The Pragmatic Programmers

TURING 图灵程序设计丛书

SQL Antipatterns

Avoiding the Pitfalls of Database Programming

SQL反模式

- 深入剖析数据库编程常见错误
- 提升SQL功力的实用宝典
- 大师指点令人茅塞顿开



[美] Bill Karwin 著
谭振林 Push Chen 译

人民邮电出版社
POSTS & TELECOM PRESS



Bill Karwin 作为软件工程师、咨询师和管理者，他在20年间开发并支持了各种各样的应用、程序库以及服务器，如PHP 5的Zend Framework、Interbase关系型数据库，以及Enhydra Java应用服务器等。他一直无私地分享他的专业知识，来帮助其他程序员提高效率，获得成功。他曾以各种方式回答了上千个关于SQL的疑问，其中不乏一些严重但又经常被忽略的问题。



谭振林 （http://weibo.com/thinhunan），资深研发人员，曾连任五届微软最有价值专家，出版多本专业译著，目前担任产品总监，在互联网产品路上孜孜探索。



Push Chen 专注于大数据量分布式存储及缓存的后台服务设计与开发，现任职于盛大在线推他项目组。

图书在版编目（CIP）数据

SQL反模式 / (美) 卡尔文 (Karwin,B.) 著 ; 谭振林译. -- 北京 : 人民邮电出版社, 2011.9
（图灵程序设计丛书）
书名原文: SQL Antipatterns:Avoiding the Pitfalls of Database Programming
ISBN 978-7-115-26127-4

Ⅰ. ①S… Ⅱ. ①卡… ②谭… Ⅲ. ①关系数据库—数据库管理系统，SQL Server Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2011)第156320号

内 容 提 要

本书是一本广受好评的 SQL 图书。它介绍了如何避免在 SQL 的使用和开发中陷入一些常见却经常被忽略的误区。它通过讲述各种具体的案例，以及开发人员和使用人员在面对这些案例时经常采用的错误解决方案，来介绍如何识别、利用这些陷阱，以及面对问题时正确的解决手段。另外，本书还涉及了 SQL 的各级范式和针对它们的正确理解。

本书适合 SQL 数据库开发人员与管理人员阅读。

图灵程序设计丛书

SQL反模式

◆ 著 [美] Bill Karwin
译 谭振林 Push Chen
责任编辑 傅志红
执行编辑 李 盼

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：16.5
字数：355千字 2011年9月第1版
印数：1 – 3 000册 2011年9月北京第1次印刷
著作权合同登记号 图字：01-2010-7681号
ISBN 978-7-115-26127-4

定价：59.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版权声明

读者感言

对于经常碰到本书中表述的那些数据库设计决择的软件开发人员来说，这本书是必读的。因为它将帮助开发团队理解数据库设计达成的结果，并且基于实际的需求、预期、测定做出最合理的决定。

——Darby Felton，DevBots Software Development 的联合创始人

我非常喜欢 Bill 在书中采用的写作方式，展示了他独一无二的风格和幽默感，这对讨论一大堆枯燥的话题太重要了。Bill 成功运用了一种很好的表述方式，让技术以易于理解的面貌示人，而且便于以后查阅。简而言之，这将是你书架里又一极好的资源。

——Arjen Lentz，Open Query
（http://openquery.com）执行总监，
High Performance MySQL 第二版作者之一

对于有 SQL 基础，但是在为项目设计 SQL 数据库时希望寻求基础之外帮助的软件工程师来说，这本书尤其有用。

——Liz Neely，资深数据库程序员

Bill 捕捉到了我们在 SQL 不同方面碰到过的很多陷阱的关键所在，而我们有些时候甚至没有意识到已被困住了。Bill 的反模式涵盖从“不敢相信我又犯了一遍”的事后感叹，到最佳方案与伴你一路走来的 SQL 教条相左的诡异情况。这是一本对 SQL 骨灰和新手都不错的书。

——Danny Thorpe，
Microsoft 总工程师；
Delphi Component Design 作者

译 者 序 一

毫无疑问，数据库领域当下最热门的概念是 NoSQL，我正在公司最新大型社区项目中实践 NoSQL 产品，并准备在更大范围内推广优秀的 NoSQL 产品，而另一译者——陈魏明，则自己研发了一个 NoSQL 产品，应用在另一大型社区项目中，提供 Feed 系统的支持。

但是，正如 NoSQL 自身所宣扬的一样，任何一种 NoSQL 产品，甚至所有的 NoSQL 产品合在一起，它们的设计初衷绝不是解决掉所有的数据处理需求，它们追求的是为某一种或某几种数据处理场景选择最优的 CAP[①]组合，提供最合适的解决方案。

因此，SQL 并没有因为 NoSQL 的流行而变得不重要，它仍然跟以前一样重要，并且因为它长期以来在开发人员中建立的深厚基础，以及丰富的支持工具，特别是强大的查询功能，将使其长期在广泛的数据处理场景中作为主要的解决方案而存在。比如你总不能用 Redis[②]来处理运营团队天天变着戏法要的运营分析数据吧。

这本语言略显啰嗦的书，是一本非常实用的书，因为它每一章的内容都源自于最常见、最普通的 SQL 应用场景，每一章中描述的问题，都是全世界的 SQL 应用人员犯得最多的错误。总之，我译完这本书后，就有一个强烈的感触："原来我犯了这么多错误！"

所以，这本书中的知识与教训，应该对很多人（比我资浅的那一小部分人和比我资深的那一大部分人）都有帮助，而且长期有效。

本书作者知识渊博，原作中不少地方引经据典，我们在翻译过程中尽量通过 Google 等办法查找对应的典故，不过文化的差异和有限的水平，造成译稿中必定还有不少不尽如人意的地方，请各位读者原谅。

谭振林

2011 年 5 月

① CAP 是指：一致性（Consistency），可用性（Availability），分区容忍性（Partition tolerance）。CAP 原理认为这三个要素最多只能同时实现两点，不可能三者兼顾。

② Redis 是一款优秀的、基于内存处理数据的、原生支持多种数据结构的 Key-Value 型 NoSQL 产品。

译者序二

我本人并非 DBA，但工作中时刻都要和数据打交道，无论是 SQL 还是 NoSQL。在一个产品的生命周期里，设计与开发总是只占很小的一部分，大量的时间都用在后续的维护、优化和调整中。互联网服务更是如此。而维护、性能调优阶段，就是一个不断地犯错—纠正—归纳的循环。本书作者将他所归纳整理出的这些“错误”写在了这本书中，让我们这些后来者能够更早地发现程序中的问题，从而节省大量的时间成本。

无论时下 NoSQL 技术有多么的火爆，终究有一个无法解决的问题——内存开销。因而几乎所有的网站、社区，其后台必然有大量使用 SQL 进行存储的模块。

在翻译本书的过程中，我也回顾了以前所接触到的或者自己做的数据库设计，很多都没有从数据库本身出发去解决问题，而是直接在前端增加了一个缓存。

本书中的很多问题解决方案都很优雅，在不触碰程序员追求程序整洁度的洁癖神经的情况下，利用 SQL 本身就有的特性和功能解决了问题。这些案例非常值得我们学习。

本书作者知识丰富，在很多方面上都有所涉及，书中提到很多文化方面的内容，翻译不到位之处，敬请指正。

Push Chen

2011 年 5 月

目　　录

第 1 章 引　言

所谓专家，就是在一个很小的领域里把所有错误都犯过了的人。

▶ 尼尔斯·玻尔

我曾经拒绝过第一份关于 SQL 的工作。

获得加州大学计算机与信息科学的本科学位后不久，我接到了一个工作邀请，来自于一位曾经在加州大学工作过的经理，我们在一次校园活动相识。他当时刚刚建立了自己的软件公司，致力于使用 shell 脚本和诸如 awk 的相关工具（诸如 Ruby、Python、PHP，甚至 Perl 等现代动态语言，在当时都还不甚流行），来开发适用于众多 UNIX 平台的数据库管理系统。这个经理邀请我是因为他需要一个人来开发一些代码，识别并且执行功能有限版本的 SQL 语言。

他说："我不需要支持完整的 SQL 语言，那样工作量太大了。我只需要支持一个 SQL 语句：SELECT。"

我在学校里并没有学过 SQL。数据库在当时并不像现在这样普遍，并且当时也没有诸如 MySQL 和 PostgreSQL 之类的开源程序。但我用 shell 脚本开发过完整的应用程序，并且了解一些语法分析的技术，也做过一些关于编译器设计和计算语言学的课程项目。因此我在想是否要接受这个工作。只解析像 SQL 这样的专业语言中单独的一类语句会有多困难呢？

我找了一份 SQL 的参考资料并且立刻意识到它不同于那些支持 `if()`、`while()`语句、变量定义、表达式以及可能还有函数调用的语言。说 SELECT 只是此类语言中的一个语句，等同于说引擎只是汽车的一部分。从字面上来看，这两句话都是正确的，但是都完全掩盖了这两个主体的复杂性和深度。我意识到仅仅为了支持 SELECT 这一个语句的执行，就必须要实现一个完整的关系型数据库管理系统以及其查询引擎的全部代码。

我拒绝了这个用 shell 脚本实现 SQL 解析与关系型数据库管理系统引擎的工作机会。那个经理并没有充分理解他的项目的复杂度，可能他并不理解什么是关系型数据库管理系统（RDBMS）。

我早期使用 SQL 的经历和普通的软件开发人员并没有什么区别，和那些从计算机科学专业毕

业的学生相比也是如此。大多数人学习 SQL 语言都是因为项目所需而不得不自学的，而不是像其他的编程语言那样从头开始认真地学习。不论是对 SQL 爱好者、专业程序员或者拥有博士学位的娴熟的研究人员，SQL 就好像是程序员的一个不经过训练就学会了的软件技能。

此后，当我了解了一些 SQL 方面的知识后，我惊讶于它和那些过程式的编程语言（诸如 C、Pascal 和 shell）或是面向对象的编程语言（诸如 C++、Java、Ruby 或 Python）都有着显著的区别。SQL 是一门*声明式的编程语言*，就像 Lisp、Haskell 或者 XSLT。SQL 使用集合（set）作为根本的数据结构，而面向对象的语言使用的是对象（object）。受过传统培训的软件开发人员被所谓的“*阻抗失配*”[1]所阻碍，因此很多程序员转而使用现成的面向对象的库，以此来避免学习如何高效地使用 SQL。

自 1992 年以来，我在工作中大量使用 SQL。我在开发应用程序的过程中要使用它，我也为 InterBase RDBMS 产品提供技术支持、开发培训课程以及文档，并且开发了 Perl 和 PHP 中用于 SQL 编程的库。我在那些网络邮件列表和新闻讨论组中回答了成千上万的问题。大量重复的问题表明程序员总是一遍又一遍地犯同样的错误。

1.1 谁需要这本书

不管你是初学者还是资深人员，这本《SQL 反模式》[2]都能帮助需要使用 SQL 的程序员更有效地使用它。我和所有不同经验层次的人交流过，他们都能从这本书中获益。

你可能已经阅读过 SQL 语法的参考资料，知道所有的 SELECT 语句的子句，并且能够用它来做一些事情了。渐渐地，你可以通过阅读别人的程序代码或者文章来提升你的 SQL 技能。但你怎么能区分优劣？怎么能确定你正在学习的是最佳方法，而不是另一个可能使你陷入困境的方法？

你可能会在本书中找到一些熟悉的话题。即使你之前已经找到了解决方案，仍可以发现新的看待问题的方法。重新审视那些广为流传的错误，将能更好地确认和巩固你对优秀范例的理解。还有一些话题可能对你来说比较新鲜，我希望你能通过阅读它们来改善自己的 SQL 编程习惯。

如果你是一个训练有素的数据库管理员，可能已经知道了如何用最好的方法来避免本书所描述的 SQL 编程中易犯的错误。然而这本书也能从软件开发人员的角度来帮助你。开发人员和 DBA 之间为项目争论是很常见的，但相互尊重和团队合作能够帮助我们更有效地工作。你可以使用本书来向你负责开发的同事解释一些好的做法，以及不这么做会有怎样的后果。

1.2 本书内容

什么是“反模式”？*反模式*是一种试图解决问题的方法，但通常会同时引发别的问题。反模

① 阻抗失配（impedance mismatch）原意为当滤波器输出阻抗 Z0 和与之端接的负载阻抗 ZL 不相等时，在该端口 A 上产生反射，引入到计算机科学中指面向对象应用程序向关系型数据库存储数据时的数据不一致问题。——译者注

② 反模式英文为 Antipattern，在 SQL 中，pattern 有时译为范式，本书中采用模式一词。——译者注

式虽以不同的形式被广泛实践，但这其中仍存在一定的共通性。人们可能独立地，或是在一个同事、一本书或者一篇文章的帮助下想出一个反模式的主意。很多面向对象的软件设计与项目管理方面的反模式都记录在 Portland Pattern Repository①中，或是记录在 1998 年出版的《反模式》②（William J. Brown 等）一书中。

本书描述了我做技术支持、做培训课程、和程序员一起开发软件以及在网络论坛上回答问题时遇到的最常见的错误。这其中很多错误我自己也犯过，除了在深夜花好几个小时找到并解决掉之外并没有更好的办法。

1.2.1 本书结构

这本书将反模式分类成如下四部分。

逻辑数据库设计反模式

在你开始编码之前，需要决定数据库里存储什么信息以及最佳的数据组织方式和内在关联方式。这包含了如何设计数据库的表、字段和关系。

物理数据库设计反模式

在知道了需要存储哪些数据之后，使用你所知的 RDBMS 技术特性尽可能高效地实现数据管理。这包含了定义表和索引，以及选择数据类型。你也要使用 SQL 的“数据定义语言”，比如 CREATE TABLE 语句。

查询反模式

你需要向数据库中添加然后获取数据。SQL 的查询是使用“数据操作语言”来完成，比如 SELECT、UPDATE 和 DELETE 语句。

应用程序开发反模式

SQL 应该会用在使用 C++、Java、Php、Python 或者 Ruby 等语言构建的应用程序中。在应用程序中使用 SQL 的方式有好有坏，该部分内容描述了一些常见错误。

很多反模式章节都采用了一些比较幽默或是能产生共鸣的标题，比如金锤子、重新造轮子或由委员会设计。通常来说，会同时给出正面的设计模式和反模式的名字作为隐喻或帮助记忆。

附录提供了一些对关系数据库理论实践的描述。本书中的很多反模式其实都是对数据库理论的误解造成的。

---

① Portland Pattern Repository：http://c2.com/cgi-bin/wiki?AntiPattern。

② 本书由人民邮电出版社于 2007 年出版。——编者注

1.2.2 反模式分解

每个反模式章节都包含了如下的子标题结构。

目的

这是你可能要去尝试解决的任务。意图使用反模式提供解决方案，但通常会以引起更多问题而告终。

反模式

这一部分表述了通常使用的解决方案的本质，并且展示了那些没有预知到的后果，正是这些使得这些方案成为反模式。

如何识别反模式

一些固定的方式会有助于你辨识在项目中使用的反模式。你遇到的特殊障碍，或是你自己和别人说的一些话，都能使你提前识别出反模式。

合理使用反模式

规则总有例外。在某些情况下，本来认为是反模式的设计却可能是合理的，或者说至少是所有的方案中最合理的。

解决方案

这一部分描述了首选的解决方案，它们不仅能够解决原有的问题，同时也不至于引起由反模式导致的新问题。

1.3 本书未涉及的内容

我不打算讲解 SQL 的语法或者相关术语。有大量关于这些基础内容的书籍或者网络资料。我假设你已经学习了足够多的 SQL 的语法来使用这门语言并且能够做好一些事情。

性能、可伸缩性以及优化对于很多开发数据驱动应用程序，特别是网络应用的设计人员来说是非常重要的。市面上也有很多和数据库开发性能相关的书籍。我推荐 *SQL Performance Tuning*[GP-3]和 *High Performance MySQL, Second Edition*[SZT+08]①。本书的一些主题和性能相关，但这不是本书最主要的目的。

我尝试把问题表述成适用于所有厂商的数据库，并且这些解决方案也将会适用于所有的数据库。SQL 语言被规定为一种 ANSI 和 ISO 标准，所有的数据库都支持这一标准，因此我尽可能中

① 《高性能 MySQL（第二版）》由电子工业出版社于 2010 年出版。——编者注

立地使用 SQL 而不偏向任何品牌，并且我会明确说明不同厂商 SQL 数据库的特定扩展。

数据访问框架和对象关系映射（ORM）库是非常有用的工具，但这些也并不是本书的重点。我用最普通、最直白的方式写过很多 PHP 的代码范例。本书的范例足够简单，从而能够在大部分编程语言中适用。

数据库的管理和操作任务，诸如服务器磁盘分配、安装和配置、监控和备份、日志分析以及安全都是非常重要并且值得用一整本书来描述的，但本书更倾向于那些使用 SQL 的开发人员而不是数据库管理员。

这本书是关于 SQL 和关系型数据库的，以及其他的替代技术，诸如面向对象数据库、键/值存储、面向列的数据库、面向文档的数据库、分级数据库、网络数据库、Map/Reduce 框架或者语义数据存储。比较这些技术的优缺点并在不同的数据管理解决方案中恰当地使用它们是一个很有趣的课题，但这并不是本书的议题。

1.4 规约

下面描述了本书中使用的一些规约。

排版

SQL 关键字都大写并且使用等宽字体，以使得它们在上下文中更突出，就像 `SELECT`。

SQL 的表名，也用等宽字体，并且首字母大写，如 `Accounts` 或者 `BugsProducts`。SQL 的列，也使用等宽字体，全都使用小写，并且使用下划线分词，如 `account_name`。

术语

SQL 的正确发音是 S-Q-L，不是 C-QL。虽然我对通俗用法并没有什么反对意见，但我更倾向于使用正式用法。

在数据库相关的用法中，“索引”一词指的是一个有序的信息集合。在其他情况下“索引”可能指的是指示器。

在 SQL 中，术语*查询*（query）和*语句*（statement）有时指的是同一个意思，指的都是任何可执行的一句完整的 SQL 指令。为了描述清晰，我使用“查询”表示 `SELECT` 语句，而“语句”用来表达其他语句，包括 `INSERT`、`UPDATE` 和 `DELETE` 以及数据定义语句。

数据实体关系图

最常见的关系数据库图表就是*实体关系图* ERD（Entity Relationship Diagram）。表格使用矩形表示，关系使用链接各个矩形的线段来表示，并且在两端使用各种符号来表述关系的基数。具体事例可以参考下一页的图 1-1。

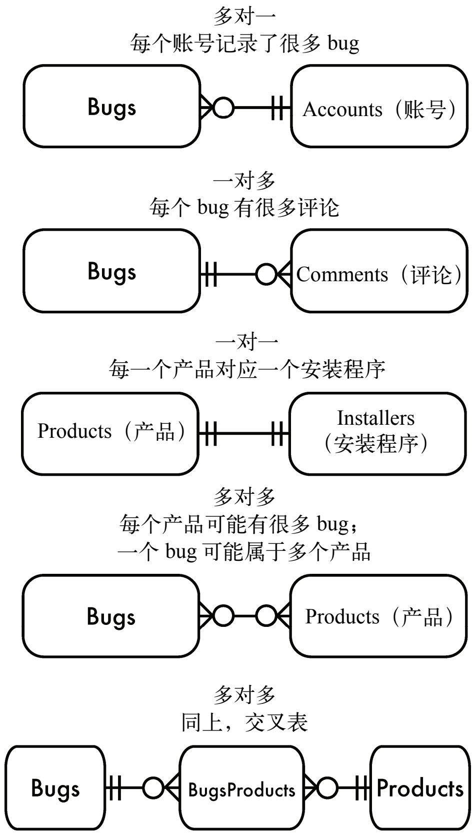


图 1-1　实体关系图 ERD 示例

1.5　范例数据库

我使用一个假想的缺陷跟踪程序的数据库来展示大部分与 SQL 反模式相关的话题。图 1-2 是该数据库的 ERD。请注意，Bugs 表和 Accounts 表之间有 3 个连接，代表 3 个不同的外键。

接下来的数据库定义语句则展示出如何定义这些表。有些时候所做的选择只是为了照顾后面的范例，因而它们可能并不是人们在实际应用程序中所做的选择。我尽力使用标准 SQL 来定义这个数据库，使其能适用于任一数据库产品，但也会出现一些 MySQL 数据类型，诸如 SERIAL 和 BIGINT 语句。

Introduction/setup.sql

```
CREATE TABLE Accounts (
  account_id        SERIAL PRIMARY KEY,
  account_name      VARCHAR(20),
  first_name        VARCHAR(20),
  last_name         VARCHAR(20),
```

```
  email              VARCHAR(100),
  password_hash      CHAR(64),
  portrait_image     BLOB,
  hourly_rate        NUMERIC(9,2)
);

CREATE TABLE BugStatus (
  status             VARCHAR(20) PRIMARY KEY
);

CREATE TABLE Bugs (
  bug_id             SERIAL PRIMARY KEY,
  date_reported      DATE NOT NULL,
  summary            VARCHAR(80),
  description        VARCHAR(1000),
  resolution         VARCHAR(1000),
  reported_by        BIGINT UNSIGNED NOT NULL,
  assigned_to        BIGINT UNSIGNED,
  verified_by        BIGINT UNSIGNED,
  status             VARCHAR(20) NOT NULL DEFAULT 'NEW',
  priority           VARCHAR(20),
  hours              NUMERIC(9,2),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);

CREATE TABLE Comments (
  comment_id         SERIAL PRIMARY KEY,
  bug_id             BIGINT UNSIGNED NOT NULL,
  author             BIGINT UNSIGNED NOT NULL,
  comment_date       DATETIME NOT NULL,
  comment            TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);

CREATE TABLE Screenshots (
  bug_id             BIGINT UNSIGNED NOT NULL,
  image_id           BIGINT UNSIGNED NOT NULL,
  screenshot_image   BLOB,
  caption            VARCHAR(100),
  PRIMARY KEY       (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

CREATE TABLE Tags (
  bug_id             BIGINT UNSIGNED NOT NULL,
  tag                VARCHAR(20) NOT NULL,
```

```
  PRIMARY KEY      (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

CREATE TABLE Products (
  product_id        SERIAL PRIMARY KEY,
  product_name      VARCHAR(50)
);

CREATE TABLE BugsProducts(
  bug_id            BIGINT UNSIGNED NOT NULL,
  product_id        BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY      (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

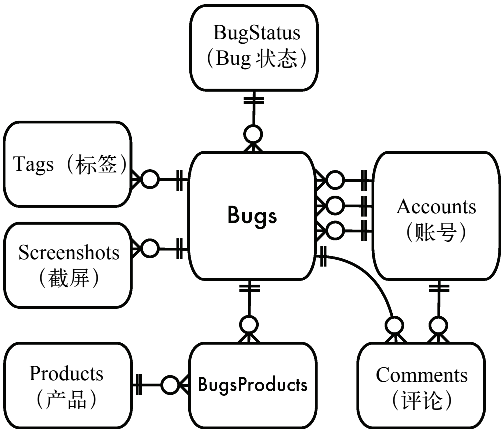


图 1-2 BUG 数据库 ERD 图

在一些章节中，尤其是在逻辑数据库反模式的章节中，我展示了一些不同的数据库定义，既为了展示反模式，也同时展示一个可选的解决方案来避免反模式。

1.6 致谢

首先，我要感谢我的妻子 Jan，没有她的启发、关爱和支持，更不用说时时的督促，我无法写成这本书。

我也想要对我的审阅者表达谢意，感谢他们花了那么多的时间。他们的建议提高了本书的质量。他们是 Marcus Adams、Jeff Bean、Frederic Daoud、Darby Felton、Arjen Lentz、Andy Lester、Chris Levesque、Mike Naberezny、Liz Nealy、Daev Roehr、Marco Romanini、Maik Schmidt、Gale Straney 以及 Danny Thorpe。

感谢编辑 Jacquelyn Carter 和 Pragmatic Bookshelf 的出版人，他们相信本书一定不辱使命。

第 2 章 乱穿马路

> 一个不愿透露姓名的 Netscape 工程师曾经将一个指针的地址当成字符串传给了 JavaScript，随后再回传给 C，节省了 30 秒。
>
> ▶ 布雷克 • 罗斯

假设你正在为缺陷跟踪程序开发一种功能，将某个用户指定为某个产品的主要联系人。你最初的设计只允许每个产品拥有一个联系人，然而，就像你猜的那样，需求可能会变为支持“每个产品有多个联系人”。

此时，将数据库中原来存储单一用户标识的字段改成使用逗号分隔的用户标识列表，似乎是一个很简单且合理的解决方案。

很快，你的老板跑来问你：“工程部在往他们的项目中添加相关人员时发现最多只能添加 5 个人，如果想要继续添加更多的人，程序就会报错，这是怎么回事？”

你点头说道：“是的，只能在一个项目中列出这么多人，就是这么设计的。”你觉得这是一件再正常不过的事情了。

但老板似乎并不这么认为，他需要一个更加明确的解释。“好吧，5 到 10 个，可能再多存几个，那取决于这些账号创建时间的早晚。”老板感觉非常吃惊，于是你继续说道：“我使用逗号分隔的列表来存储项目的账号 ID，而这个 ID 列表的长度不能超过字符串的最大长度值。账号 ID 越短，列表中的账号 ID 就越多。因此，账号创建较早的人，他们的 ID 不大于 99，这些账号 ID 更短。”

老板皱起眉头，你觉得你又要加班到很晚了。

程序员通常使用逗号分隔的列表来避免在多对多的关系中创建交叉表，我将这种设计方式定义为一种反模式，称为*乱穿马路*（Jaywalking），因为乱穿马路也是避免过十字路口的一种方式。

2.1　目标：存储多值属性

设计一个单值表列是非常简单的：你可以选择一个 SQL 的内置数据类型，以该类型来存储这个表项的数据，比如整型、日期类型或者字符串。但是如何才能做到在一列中存储一系列相关数据的集合呢？

在我们的缺陷跟踪数据库的例子中，我们在 Products 表中使用一个整型的列来关联产品和对应的联系人。每个账号可能对应很多产品，每个产品又引用了一个联系人，因此我们在产品和账号之间有一个多对一的关系。

Jaywalking/obj/create.sql

```
CREATE TABLE Products (
  product_id   SERIAL PRIMARY KEY,
  product_name VARCHAR(1000),
  account_id   BIGINT UNSIGNED,
  -- . . .
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', 12);
```

随着项目日趋成熟，你意识到一个产品可能会有多个联系人。除了多对一的关系之外，我们还需要支持产品到账号的一对多的关系。Products 表中的一行数据必须要能够存储多个联系人。

2.2　反模式：格式化的逗号分隔列表

为了将对数据库表结构的改动控制在最小范围内，你决定将 account_id 的类型修改成 VARCHAR，这样就可以列出该列中的多个账号 ID，每个账号 ID 之间用逗号分隔。

Jaywalking/anti/create.sql

```
CREATE TABLE Products (
  product_id   SERIAL PRIMARY KEY,
  product_name VARCHAR(1000),
  account_id   VARCHAR(100), -- 逗号分隔列表
  -- . . .
);
```

这样的设计似乎可行，因为你没有创建额外的表或者列，而仅仅改变了一个字段的数据类型。然而，我们来看一下这样设计所必须承受的性能问题和数据完整性问题。

2.2.1　查询指定账号的产品

如果所有的外键都合并在一个单元格内，查询会变得异常困难。你将不能再使用等号，相反，

不得不对某类模式使用测试。比如，MySQL 下可以写一些如下的表达式来查询所有账号 ID 为 12 的产品：

Jaywalking/anti/regexp.sql

```
SELECT * FROM Products WHERE account_id REGEXP '[[:<:]]12[[:>:]]';
```

模式匹配的表达式可能会返回错误结果，并且无法享受索引带来的性能优势。由于模式匹配表达式的语法在不同品牌的数据库中是不同的，因此你的 SQL 代码并不是平台中立的。

2.2.2 查询指定产品的账号

同样地，使用逗号分隔的列表来做多表联结查询定位一行数据也是极不优雅和耗时的。

Jaywalking/anti/regexp.sql

```
SELECT * FROM Products AS p JOIN Accounts AS a
    ON p.account_id REGEXP '[[:<:]]' || a.account_id || '[[:>:]]'
WHERE p.product_id = 123;
```

联合两张表并使用如上的一句表达式将毁掉任何使用索引的可能。这样的查询必须扫描两张表，创建一个交叉结果集，然后使用正则表达式遍历每一行联合后的数据进行匹配。

2.2.3 执行聚合查询

聚合查询使用 SQL 内置的聚合函数，如 COUNT()、SUM()、AVG()。然而，这些函数是针对分组行而设计的，并不是为了逗号分隔的列表。你不得不借助于如下的一些方法：

Jaywalking/anti/count.sql

```
SELECT product_id, LENGTH(account_id) - LENGTH(REPLACE(account_id, ',', '')) + 1
    AS contacts_per_product
FROM Products;
```

这类方法可能看上去很高明，但并不清晰。这种类型的解决方案需要花费很长的时间来开发并且不方便调试。何况有些聚合查询根本无法使用这些技巧来完成。

2.2.4 更新指定产品的账号

你可以用字符串拼接的方式在列表尾端增加一个新的 ID，但这并不能使列表按顺序存储。

Jaywalking/anti/update.sql

```
UPDATE Products
SET account_id = account_id || ',' || 56
WHERE product_id = 123;
```

要从列表中删除一个条目，必须执行两条 SQL 查询语句：第一条提取老的列表，第二条存储更新后的列表。

Jaywalking/anti/remove.php

```
<?php

$stmt = $pdo->query(
  "SELECT account_id FROM Products WHERE product_id = 123");
$row = $stmt->fetch();
$contact_list = $row['account_id'];

// change list in PHP code
$value_to_remove = "34";
$contact_list = split(",", $contact_list);
$key_to_remove = array_search($value_to_remove, $contact_list);
unset($contact_list[$key_to_remove]);
$contact_list = join(",", $contact_list);

$stmt = $pdo->prepare(
  "UPDATE Products SET account_id = ?
   WHERE product_id = 123");
$stmt->execute(array($contact_list));
```

仅仅为了从列表中删除一个账号就要写如此多的代码。

2.2.5 验证产品ID

用什么来防止用户在 ID 中输入诸如“banana”（香蕉）这样的非法字段？

Jaywalking/anti/banana.sql

```
INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', '12,34,banana');
```

用户总能找到办法输入他们想输入的东西，然后你的数据库就会变得很乱。上面这样的情况并不一定是一个数据库错误，但相对应的数据会变得毫无价值。

2.2.6 选择合适的分隔符

如果存储一个字符串列表而不是数字列表，列表中的某些条目可能会包含分隔符。使用逗号作为分隔符可能会有问题。当然，你到时候可以再换一个字符，但你能确保这个新字符永远不会出现在条目中吗？

2.2.7 列表长度限制

你能在一个 VARCHAR(30)的结构中存多少数据呢？这依赖于每个条目的长度。如果每个条目只有 2 个字符长，那你能存 10 个条目（包括逗号）。但如果每个条目的长度为 6，你就只能存 4 个了。

Jaywalking/anti/length.sql

```
UPDATE Products SET account_id = '10,14,18,22,26,30,34,38,42,46'
WHERE product_id = 123;

UPDATE Products SET account_id = '101418,222630,343842,467790'
WHERE product_id = 123;
```

你怎能确定VARCHAR(30)能够支持你未来所需的最长列表呢？多长才够长？你可以自己尝试着去和老板或者客户解释这么限制的原因。

2.3 如何识别反模式

如果你的项目团队说过下面这些话，那么这很有可能就是在项目中使用了“乱穿马路”设计模式的线索。

- “列表最多支持存放多少数据？”
  这个问题在选择 VARCHAR 列的最大长度时被提及。
- “你知道在 SQL 中如何做分词查找吗？”
  如果你用了正则表达式来提取字符串中的组成部分，这可能是一种提示，意味着你应该把这些数据分开存储。
- “哪些字符不会出现在任何一个列表条目中？”
  你想要使用一个不会令人困惑的分割符号，但你也应该明白任何字符都有可能在某天出现在字段中的某个值内。

2.4 合理使用反模式

出于性能优化的考量，可能在数据库的结构中需要使用反规范化的设计。将列表存储为以逗号分隔的字符串就是反规范化的一个实例。

应用程序可能会需要逗号分隔的这种存储格式，也可能没必要获取列表中的单独项。同样，如果应用程序接收的源数据是有逗号分隔的格式，而你只需要存储和使用它们并且不对其做任何修改，完全没必要分开其中的值。

你需要谨慎使用反规范化的数据库设计。尽可能地使用规范化的数据库设计，因为那样的设计能让你的产品代码更灵活，并且也能在数据库层保持数据完整性。

2.5 解决方案：创建一张交叉表

将 account_id 存储在一张单独的表中，而不是存储在 Products 表中，从而每个独立的

account 值都可以占据一行。这张新表称为 Contacts，实现了 Products 和 Accounts 的多对多关系。

Jaywalking/soln/create.sql

```
CREATE TABLE Contacts (
  product_id  BIGINT UNSIGNED NOT NULL,
  account_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (product_id, account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id),
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Contacts (product_id, accont_id)
VALUES (123, 12), (123, 34), (345, 23), (567, 12), (567, 34);
```

当一张表有指向另外两张表的外键时，我们称这种表为一张*交叉表*[①]，它实现了两张表之间的多对多关系。这意味着每个产品都可以通过交叉表和多个账号关联；同样地，一个账号也可以通过交叉表和多个产品关联。参考图 2-1。



图 2-1　交叉表实体关系图

我们来看看，使用交叉表是如何解决 2.2 节中描述的问题的。

2.5.1　通过账号查询产品和反过来查询

要查询指定账号的产品的所有属性，使用 Products 和 Contacts 表的联结查询是再简单不过的了：

Jaywalking/soln/join.sql

```
SELECT p.*
FROM  Products AS p JOIN Contacts AS c ON (p.account_id = c.account_id)
WHERE c.account_id = 34;
```

有些人拒绝使用联结查询，他们认为那样很低效。然而，与 2.2 节中提出的方法相比，这个查询更好地使用了索引。

查询账号的细节也非常地简单，也方便优化。这里使用索引提高了联结查询的效率，而不是使用深奥的正则表达式：

① 有人用“联合表”、“多对多表”、“映射表”或其他术语来描述这种表，表述方式不同而已，其本质都是相同的。——译者注

Jaywalking/soln/join.sql

```
SELECT a.*
FROM   Accounts AS a JOIN Contacts AS c ON (a.account_id = c.account_id)
WHERE c.product_id = 123;
```

2.5.2 执行聚合查询

下面的例子返回每个产品相关的账号数量：

Jaywalking/soln/group.sql

```
SELECT product_id, COUNT(*) AS accounts_per_product
FROM Contacts
GROUP BY product_id;
```

获取每个账号相关的产品数量也很简单：

Jaywalking/soln/group.sql

```
SELECT account_id, COUNT(*) AS products_per_account
FROM Contacts
GROUP BY account_id;
```

生成其他更加复杂的报表也是可行的，比如找到相关账号最多的产品：

Jaywalking/soln/group.sql

```
SELECT c.product_id, c.accounts_per_product
FROM (
  SELECT product_id, COUNT(*) AS accounts_per_product
  FROM Contacts
  GROUP BY product_id
) AS c
HAVING c.accounts_per_product = MAX(c.accounts_per_product)
```

2.5.3 更新指定产品的相关联系人

你可以通过添加或者删除交叉表中的行来简单地修改字段中的条目。在 Contacts 表中，每条产品记录都是分开存储在不同的行中的，因而可以添加或删除。

Jaywalking/soln/remove.sql

```
INSERT INTO Contacts (product_id, account_id) VALUES (456, 34);

DELETE FROM Contacts WHERE product_id = 456 AND account_id = 34;
```

2.5.4 验证产品ID

你可以以另一张表中的合法数据为标准，使用外键来验证数据。通过声明 Contacts.account_id 引用 Accounts.account_id，就能依靠数据库自身的约束来强制引用的完整性，以

确保交叉表中只包含确凿存在的账号 ID。

你还可以使用 SQL 的数据类型来约束条目。例如，设定字段中的条目应该是 INTEGER 或者 DATE 类型的，就可以确信所有条目都是合法的数据（不会是乱七八糟的像“banana”一样的数据）。

2.5.5 选择分隔符

你用不到分隔符了，因为数据都分开存储在不同的行中。即使条目中出现了逗号或者任何你可能想用做分隔符的其他字符，也不会有任何问题。

2.5.6 列表长度限制

至此，每个条目都位于交叉表中的独立的行内，列表的长度限制就变成了一张表可以实际存放的行数。如果可以限制条目总数，你应该在程序中加强对条目数量的使用，而不是统计列表的总体长度。

2.5.7 其他使用交叉表的好处

为 Contacts.account_id 做索引的查询效率比用逗号分隔列表中分串高效得多。在许多数据库中，声明某一列为外键会隐式地为该列创建索引（实际情况以相关文档为准）。

你还可以在交叉表中添加其他属性。比如，可以记录一个联系人被加入产品的具体日期，或产品的第一联系人及第二联系人。这些都是在逗号分隔的列表中无法做到的。

每个值都应该存储在各自的行与列中。

第 3 章 单纯的树

树就是树，你还需要考虑些什么呢？

▶ 罗纳德·里根

设想你正在为一个著名的科技新闻网站开发新版本。

这是一个很前卫的网站，因此，读者可以评论原文甚至相互回复，这样就某一主题的讨论又延伸出很多新的分支，其深度就会大大增加。你选择了一个简单的解决方案来跟踪这些回复分支：每条评论引用它所回复的评论。

Trees/intro/parent.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  parent_id    BIGINT UNSIGNED,
  comment      TEXT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id)
);
```

很快，程序的逻辑就变得清晰起来，然而，要用一条简单的 SQL 语句检索一个很长的回复分支，还是很困难的。你只能获取一条评论的下一级或者联结第二级，到一个固定的深度。但这个贴子可以是*无限深的*，你可能需要执行很多次 SQL 查询才能获取给定主题的所有评论。

另一个你想到的主意是先获取一个主题的所有评论，然后再在程序的栈内存中将这些数据整合成你在学校里学到的传统的树形数据结构。但是，网站的产品人员告诉你，他们每天会发布数十篇文章，每篇文章可能有几百上千条评论，因此当每次有人访问网站都要做一次数据重整是不切实际的。

一定有一个更好的方法来存储评论的分支，同时可以简单而高效地获取一个完整的评论分支。

3.1 目标：分层存储与查询

存在递归关系的数据很常见，数据常会像树或者以层级方式组织。在树形结构中，实例被

称为节点（node）。每个节点有多个子节点和一个父节点。最上层的节点叫根（root）节点，它没有父节点。最底层的没有子节点的节点叫叶（leaf）。而中间的节点简单地称为非叶（nonleaf）节点。

在层级数据中，你可能需要查询与整个集合或其子集相关的特定对象，例如下述树形数据结构。

组织架构图。职员与经理的关系是典型的树形结构数据，出现在无数的 SQL 书籍与论题中。在组织架构图中，每个职员有一个经理，在树结构中表现为职员的父节点。同时，经理也是一个职员。

话题型讨论。正如引言中介绍的，树形结构也能用来表示回复评论的评论链。在这种树中，评论的子节点是它的回复。

本章将用话题型讨论作为案例来讨论反模式及其解决方案。

3.2　反模式：总是依赖父节点

在很多书籍或文章中，最常见的简单解决方案是添加 `parent_id` 字段，引用同一张表中的其他回复。可以建一个外键约束来维护这种关系。下面是表的定义，图 3-1 是实例关系图。

Trees/anti/adjacency-list.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  parent_id    BIGINT UNSIGNED,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

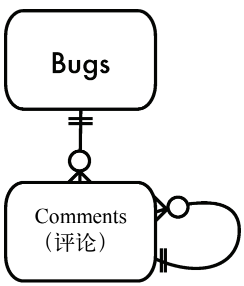


图 3-1　邻接表的 ERD

这样的设计叫做邻接表。这可能是程序员们用来存储分层结构数据中最普通的方案了。下表展示了一些简单的范例数据来显示评论表中的分层结构数据，同时图 3-2 展示了一棵该结构的树。

comment_id	parent_id	author	comment
1	NULL	Fran	这个Bug的成因是什么
2	1	Ollie	我觉得是一个空指针
3	2	Fran	不，我查过了
4	1	Kukla	我们需要查无效输入
5	4	Ollie	是的，那是个问题
6	4	Fran	好，查一下吧
7	6	Kukla	解决了

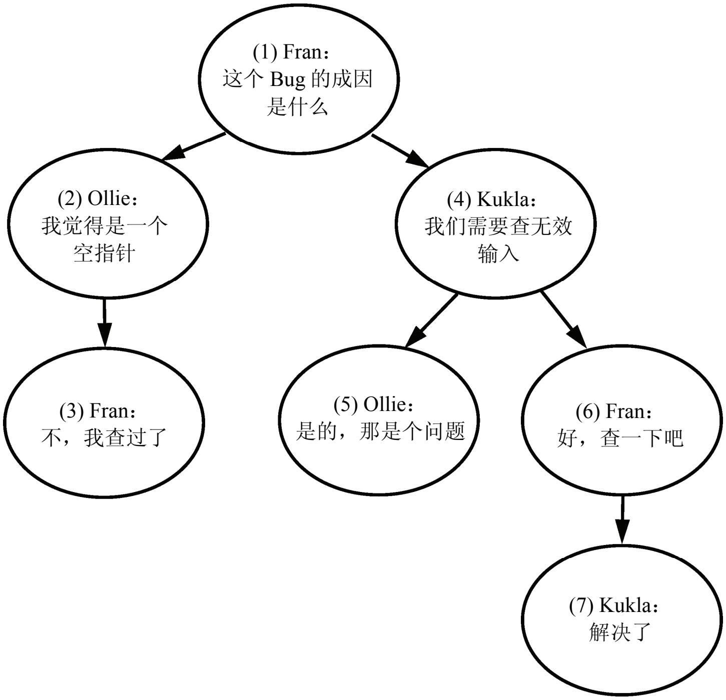


图 3-2 线程化讨论示意图

3.2.1 使用邻接表查询树

但是，就算如此多的程序员会将邻接表作为默认的解决方案，它仍然可能成为一个反模式，原因在于它无法完成树操作中最普通的一项：查询一个节点的所有后代。你可以使用一个关联查询来获取一条评论和它的直接后代：

Trees/anti/parent.sql

```
SELECT c1.*, c2.*
FROM Comments c1 LEFT OUTER JOIN Comments c2
  ON c2.parent_id = c1.comment_id;
```

然而，这个查询只能获取两层的数据。树的特性就是它可以任意深地扩展，因而你需要有方法来获取任意深度的数据。比如，可能需要计算一个评论分支的数量，或者计算一个机械设备所有部分的总开销。

当你使用邻接表的时候，这样的查询会变得很不优雅，因为每增加一层的查询都会需要额外扩展一个联结，而 SQL 查询中联结的次数是有上限的。如下的查询能够获得四层数据，但无法更多了：

Trees/anti/ancestors.sql

```
SELECT c1.*, c2.*, c3.*, c4.*
FROM Comments c1                     -- 1st level
  LEFT OUTER JOIN Comments c2
    ON c2.parent_id = c1.comment_id  -- 2nd level
LEFT OUTER JOIN Comments c3
  ON c3.parent_id = c2.comment_id  -- 3rd level
LEFT OUTER JOIN Comments c4
  ON c4.parent_id = c3.comment_id; -- 4th level
```

这样的查询很笨拙，因为伴随着逐渐增加的后代层次，必须同等地增加联结查询的列。这使得执行一个聚合函数（比如 COUNT()）变得极其困难。

另一种通过邻接表来获取树结构数据的方法是，先查询出所有行，在应用程序中自顶向下地重新构造出这棵树，然后再像树一样使用这些数据。

Trees/anti/all-comments.sql

```
SELECT * FROM Comments WHERE bug_id = 1234;
```

在处理数据之前就进行从数据库到应用程序之间的大量数据复制，是非常低效的，你可能仅仅需要一棵子树，而不是从根开始的完整的树；或者可能仅仅需要这些数据的聚合信息，比如评论的总数量（使用 COUNT()函数）。

3.2.2 使用邻接表维护树

无可否认，一些操作通过邻接表来实现是非常方便的，比如增加一个叶子节点：

Trees/anti/insert.sql

```
INSERT INTO Comments (bug_id, parent_id, author, comment)
  VALUES (1234, 7, 'Kukla', 'Thanks!');
```

修改一个节点的位置或者一棵子树的位置也是很简单的：

Trees/anti/update.sql

```
UPDATE Comments SET parent_id = 3 WHERE comment_id = 6;
```

然而，从一棵树中删除一个节点会变得比较复杂。如果需要删除一棵子树，你不得不执行多次查询来找到所有的后代节点，然后逐个从最低级别开始删除这些节点以满足外键完整性。

Trees/anti/delete-subtree.sql

```
SELECT comment_id FROM Comments WHERE parent_id = 4; -- returns 5 and 6
SELECT comment_id FROM Comments WHERE parent_id = 5; -- returns none
SELECT comment_id FROM Comments WHERE parent_id = 6; -- returns 7
SELECT comment_id FROM Comments WHERE parent_id = 7; -- returns none

DELETE FROM Comments WHERE comment_id IN ( 7 );
DELETE FROM Comments WHERE comment_id IN ( 5, 6 );
DELETE FROM Comments WHERE comment_id = 4;
```

可以使用一个带 ON DELETE CASCADE 修饰符的外键约束来自动完成这些操作，只要能肯定确实是要删除这些节点，而不是修改它们的位置。

假如想要删除一个非叶子节点并且提升它的子节点，或者将它的子节点移动到另一个节点下，那么首先要修改子节点的 parent_id，然后才能删除那个节点。

Trees/anti/delete-non-leaf.sql

```
SELECT parent_id FROM Comments WHERE comment_id = 6; -- returns 4

UPDATE Comments SET parent_id = 4 WHERE parent_id = 6;

DELETE FROM Comments WHERE comment_id = 6;
```

这些都是使用邻接表时需要多步操作才能完成的查询范例，你不得不写很多额外的代码，而其实数据库设计本身就能做得很简单和高效。

3.3　如何识别反模式

如果听到了如下的问题，可能你正在使用“单纯的树”这种反模式。

- “我们的树结构要支持多少层？”
  你正纠结于不使用递归查询获取一个给定节点的所有祖先或者后代。你做出让步，只支持有限层级数据的所有的操作，然后紧接着的一个很自然的问题就是：多少层才足够满足需求？
- “我总是很怕接触那些管理树结构的代码。”
  你已经采用了一种管理分层数据的比较复杂的方法，但使用的是错误的方法。每一项技术都会使得一些任务变得很简单，但通常是以其他的任务变得更复杂作为代价的。你可能选择了在应用程序设计中并不是最佳的方案来管理这些分层数据。

- “我需要一个脚本来定期地清理树中的孤立节点数据。”
  你的应用程序因为删除了非叶子节点而产生了一些迷失节点。在数据库中存储了一些复杂的结构时，需要确保在任何改变之后，这个结构都处在一致的、有效的状态下。你可以使用本章后面所描述的任何解决方案，同时配合触发器、级联外键等，来使得数据存储的结构更加适合项目需求，尽可能少地产生零碎的数据。

3.4 合理使用反模式

某些情况下，在应用程序中使用邻接表设计可能正好适用。邻接表设计的优势在于能快速地获取一个给定节点的直接父子节点，它也很容易插入新节点。如果这样的需求就是你的应用程序对于分层数据的全部操作，那使用邻接表就可以很好地工作了。

不要过度设计

我曾经为一个计算机数据中心写过一个库存跟踪程序。一些器材安装在电脑主机内部。比如缓存磁盘控制器是装在一个机架服务器里面的，扩展内存模块是装在磁盘控制器上的，等等。

我需要一个简单的数据库解决方案，来追踪那些包含了其他小部件的大设备的使用情况，同时也需要追踪每个独立部件的情况，并给出关于设备使用情况、折旧状态以及投资收益率的报表。

项目经理认为大的部件包含了一系列其他部件，同时这些部件还能再包含更小的部件，因而数据结构上这种层级关系可以是任意深度的。最终那花了我几个星期的时间来调整代码，以便让这棵树很好地适应数据库、用户界面、管理员界面和最终生成的报表。

然而在实际生产过程中，这个库存系统从来不曾有哪些设备间的关系达到两层嵌套，都是简单的父–子关系而已。如果我的最终用户能在一开始就确认两层的关系足够满足产品的需求，那么我们可以减少很大一部分的工作量。

某些品牌的数据库管理系统提供扩展的 SQL 语句，来支持在邻接表中存储分层数据结构。SQL-99 标准定义了递归查询的表达式规范，使用 WITH 关键字加上公共表表达式。

Trees/legit/cte.sql

```
WITH CommentTree
    (comment_id, bug_id, parent_id, author, comment, depth)
AS (
    SELECT *, 0 AS depth FROM Comments
    WHERE parent_id IS NULL
  UNION ALL
    SELECT c.*, ct.depth+1 AS depth FROM CommentTree ct
```

```
    JOIN Comments c ON (ct.comment_id = c.parent_id)
)
SELECT * FROM CommentTree WHERE bug_id = 1234;
```

Microsoft SQL Server 2005、Oracle 11*g*R2、IBM DB2 和 PostgreSQL 8.4 都支持使用如上的查询表达式来进行递归查询。

和 Oracle 10*g* 一样，MySQL、SQLite 和 Infomix 是少数几个暂时还不支持这种表达式的数据库，它们都被广泛地应用。也许我们可以假设将来递归查询语法将被所有的主流数据库所支持，那么使用邻接表的设计也不会再受到这么多限制了。

Oracle 9*i* 和 10*g* 也支持 WITH 子句，但并不是在递归查询时使用的。它们有着自己特殊的语法定义：START WITH 和 CONNECT BY PRIOR。

Trees/legit/connect-by.sql

```
SELECT * FROM Comments
START WITH comment_id = 9876
CONNECT BY PRIOR parent_id = comment_id;
```

3.5　解决方案：使用其他树模型

有几种方案可以代替邻接表模型，包括路径枚举、嵌套集以及闭包表。接下来我将分三段来展示这三种设计方案是如何解决 3.2 节中所描述的存储和查询树型评论的问题的。

这些解决方案通常是这样的：首先可能看上去比邻接表复杂很多，但它们的确使得某些使用邻接表比较复杂或者很低效的操作变得更简单。如果你的应用程序确实需要提供这些操作，那么这些设计会是比邻接表更好的选择。

3.5.1　路径枚举

邻接表的缺点之一是从树中获取一个给定节点的所有祖先的开销很大。路径枚举的设计通过将所有祖先的信息联合成一个字符串，并保存为每个节点的一个属性，很巧妙地解决了这个问题。

路径枚举是一个由连续的直接层级关系组成的完整路径。如/usr/local/lib 的 UNIX 路径是文件系统的一个路径枚举，其中 usr 是 local 的父亲，这也就意味着 usr 是 lib 的祖先。

在 Comments 表中，我们使用类型为 VARCHAR 的 path 字段来代替原来的 parent_id 字段。这个 path 字段所存储的内容为当前节点的最顶层的祖先到它自己的序列，就像 UNIX 的路径一样，你甚至可以使用‘/’作为路径中的分割符。

Trees/soln/path-enum/create-table.sql

```
CREATE TABLE Comments (
```

```
  comment_id   SERIAL PRIMARY KEY,
  path         VARCHAR(1000),
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

comment_id	path	author	comment
1	1/	Fran	这个Bug的成因是什么
2	1/2/	Ollie	我觉得是一个空指针
3	1/2/3/	Fran	不，我查过了
4	1/4/	Kukla	我们需要查无效输入
5	1/4/5/	Ollie	是的，那是个问题
6	1/4/6/	Fran	好，查一下吧
7	1/4/6/7/	Kukla	解决了

你可以通过比较每个节点的路径来查询一个节点的祖先。比如，要找到评论#7——路径是1/4/6/7——的祖先，可以这样做：

Trees/soln/path-enum/ancestors.sql

```
SELECT *
FROM Comments AS c
WHERE '1/4/6/7/' LIKE c.path || '%';
```

这句查询语句会匹配到路径为 1/4/6/%、1/4/%以及 1/%的节点，而这些节点就是评论#7 的祖先。

同时还可以通过将 LIKE 关键字两边的参数互换，来查询一个给定节点的所有后代。比如查找评论#4——路径为 1/4——的所有后代，可以使用如下的语句：

Trees/soln/path-enum/descendants.sql

```
SELECT *
FROM Comments AS c
WHERE c.path LIKE '1/4/' || '%';
```

这句查询语句所找到的后代的路径分别是 1/4/5、1/4/6 以及 1/4/6/7。

一旦你可以很简单地获取一棵子树或者从子孙节点到祖先节点的路径，就可以很简单地实现更多的查询，比如计算一棵子树所有节点上的值的总和（使用 SUM 聚合函数），或者仅仅是单纯地计算节点的数量。如果要计算从评论#4 扩展出的所有评论中每个用户的评论数量，可以这样做：

Trees/soln/path-enum/count.sql

```
SELECT COUNT(*)
FROM Comments AS c
```

```
WHERE c.path LIKE '1/4/' || '%'
GROUP BY c.author;
```

插入一个节点也可以像使用邻接表一样地简单。可以插入一个叶子节点而不用修改任何其他的行。你所需要做的只是复制一份要插入节点的逻辑上的父亲节点的路径，并将这个新节点的 ID 追加到路径末尾就行了。如果这个 ID 是在插入时自动生成的，你可能需要先插入这条记录，然后获取这条记录的 ID，并更新它的路径。比如，你使用的是 MySQL，它的内置函数 LAST_INSERT_ID()会返回当前会话的最新一条插入记录的 ID，通过调用这个函数，便可以获得你所需要的 ID，然后就可以通过新节点的父亲节点来获取完整的路径了。

Trees/soln/path-enum/insert.sql

```
INSERT INTO Comments (author, comment) VALUES ('Ollie', 'Good job!');

UPDATE Comments
  SET path = (SELECT path FROM Comments WHERE comment_id = 7)
    || LAST_INSERT_ID() || '/'
WHERE comment_id = LAST_INSERT_ID();
```

路径枚举的方案也存在这一些缺点，比如就存在第 2 章“乱穿马路”中所描述的缺点：数据库不能确保路径的格式总是正确或者路径中的节点确实存在。依赖于应用程序的逻辑代码来维护路径的字符串，并且验证字符串的正确性的开销很大。无论将 VARCHAR 的长度设定为多大，依旧存在长度限制，因而并不能够支持树结构的无限扩展。

路径枚举的设计方式能够很方便地根据节点的层级排序，因为路径中分隔符两边的节点间的距离永远是 1，因此通过比较字符串长度就能知道层级的深浅。

3.5.2 嵌套集

嵌套集解决方案是存储子孙节点的相关信息，而不是节点的直接祖先。我们使用两个数字来编码每个节点，从而表示这一信息，可以将这两个数字称为 nsleft 和 nsright。

Trees/soln/nested-sets/create-table.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  nsleft       INTEGER NOT NULL,
  nsright      INTEGER NOT NULL,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs (bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

每个节点通过如下的方式确定 nsleft 和 nsright 的值：nsleft 的数值小于该节点所有后

代的 ID，同时 nsright 的值大于该节点所有后代的 ID。这些数字和 comment_id 的值并没有任何关联。

确定这三个值（nsleft，comment_id，nsrigh）的简单方法是对树进行一次深度优先遍历，在逐层深入的过程中依次递增地分配 nsleft 的值，并在返回时依次递增地分配 nsright 的值。

通过图 3-3 可以简单地想象出这样的分配方式。

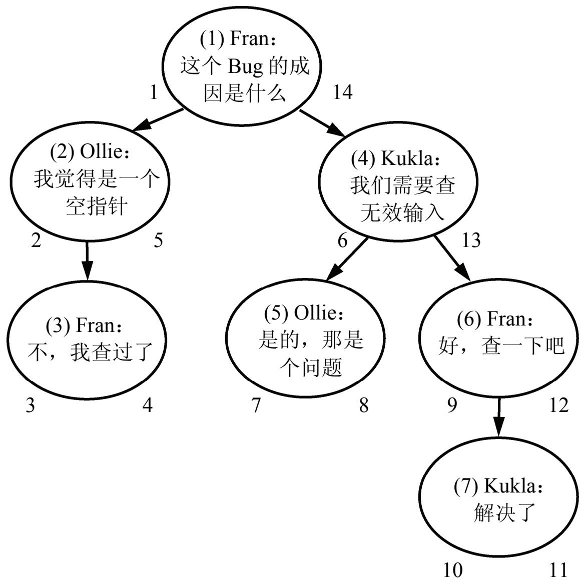


图 3-3　嵌套集示意图

comment_id	nsleft	nsright	author	comment
1	1	14	Fran	这个Bug的成因是什么
2	2	5	Ollie	我觉得是一个空指针
3	3	4	Fran	不，我查过了
4	6	13	Kukla	我们需要查无效输入
5	7	8	Ollie	是的，那是个问题
6	9	12	Fran	好，查一下吧
7	10	11	Kukla	解决了

一旦你为每个节点分配了这些数字，就可以使用它们来找到给定节点的祖先和后代。比如，可以通过搜索哪些节点的 ID 在评论#4 的 nsleft 和 nsright 范围之间来获取评论#4 及其所有后代。

Trees/soln/nested-sets/descendants.sql

```
SELECT c2.*
FROM Comments AS c1
  JOIN Comments as c2
```

```
    ON c2.nsleft BETWEEN c1.nsleft AND c1.nsright
WHERE c1.comment_id = 4;
```

通过搜索评论#6 的 ID 在哪些节点的 nsleft 和 nsright 范围之内，可以获取评论#6 及其所有祖先：

Trees/soln/nested-sets/ancestors.sql

```
SELECT c2.*
FROM Comments AS c1
  JOIN Comment AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 6;
```

使用嵌套集设计的主要优势便是，当你想要删除一个非叶子节点时，它的后代会自动地代替被删除的节点，成为其直接祖先节点的直接后代。尽管每个节点的左右两个值在示例图中是有序分配，而每个节点也总是和它相邻的父兄节点进行比较，但嵌套集设计并不必须保存分层关系。因而当删除一个节点造成数值不连续时，并不会对树的结构产生任何影响。

比如，你可以计算给定节点的深度然后删除它的父亲节点，随后，当你再次计算这个节点的深度的时候，它已经自动减少了一层。

Trees/soln/nested-sets/depth.sql

```
-- Reports depth = 3
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
FROM Comment AS c1
  JOIN Comment AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;

DELETE FROM Comment WHERE comment_id = 6;

-- Reports depth = 2
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
FROM Comment AS c1
  JOIN Comment AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;
```

然而，某些在邻接表的设计中表现得很简单的查询，比如获取一个节点的直接父亲或者直接后代，在嵌套集的设计中会变得比较复杂。在嵌套集中，如果需要查询一个节点的直接父亲，我们会这么做：给定节点 c1 的直接父亲是这个节点的一个祖先，且这两个节点之间不应该有任何其他的节点，因此，你可以用一个递归的外联结来查询一个节点 x，它即是 c1 的祖先，也同时是另一个 Y 节点的后代，随后我们使 Y=x 并继续查询，直到查询返回空，即不存在这样的节点，此时的 Y 便是 c1 的直接父亲节点。

比如，要找到评论#6的直接父亲，你可以这样做：

Trees/soln/nested-sets/parent.sql

```
SELECT parent.*
FROM Comment AS c
  JOIN Comment AS parent
    ON c.nsleft BETWEEN parent.nsleft AND parent.nsright
  LEFT OUTER JOIN Comment AS in_between
    ON c.nsleft BETWEEN in_between.nsleft AND in_between.nsright
    AND in_between.nsleft BETWEEN parent.nsleft AND parent.nsright
WHERE c.comment_id = 6
  AND in_between.comment_id IS NULL;
```

对树进行操作，比如插入和移动节点，使用嵌套集会比其他的设计复杂很多。当插入一个新节点时，你需要重新计算新插入节点的相邻兄弟节点、祖先节点和它祖先节点的兄弟，来确保它们的左右值都比这个新节点的左值大。同时，如果这个新节点是一个非叶子节点，你还要检查它的子孙节点。假设新插入的节点是一个叶子节点，如下的语句可以更新每个需要更新的地方：

Trees/soln/nested-sets/insert.sql

```
-- make space for NS values 8 and 9
UPDATE Comment
  SET nsleft = CASE WHEN nsleft >= 8 THEN nsleft+2 ELSE nsleft END,
      nsright = nsright+2
WHERE nsright >= 7;

-- create new child of comment #5, occupying NS values 8 and 9
INSERT INTO Comment (nsleft, nsright, author, comment)
  VALUES (8, 9, 'Fran', 'Me too!');
```

如果简单快速地查询是整个程序中最重要的部分，嵌套集是最佳选择——比操作单独的节点要方便快捷很多。然而，嵌套集的插入和移动节点是比较复杂的，因为需要重新分配左右值，如果你的应用程序需要频繁的插入、删除节点，那么嵌套集可能并不适合。

3.5.3　闭包表

闭包表是解决分级存储的一个简单而优雅的解决方案，它记录了树中所有节点间的关系，而不仅仅只有那些直接的父子关系。

在设计评论系统时，我们额外创建了一张叫做 TreePaths 的表，它包含两列，每一列都是一个指向 Comments 中 comment_id 的外键。

Trees/soln/closure-table/create-table.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
```

```
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);

CREATE TABLE TreePaths (
  ancestor    BIGINT UNSIGNED NOT NULL,
  descendant  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY(ancestor, descendant),
  FOREIGN KEY (ancestor) REFERENCES Comments(comment_id),
  FOREIGN KEY (descendant) REFERENCES Comments(comment_id)
);
```

我们不再使用 Comments 表来存储树的结构，而是将树中任何具有祖先–后代关系的节点对都存储在 TreePaths 表的一行中，即使这两个节点之间不是直接的父子关系；同时，我们还增加一行指向节点自己。更形象的表示可以参考图 3-4。

祖 先	后 代	祖 先	后 代	祖 先	后 代
1	1	1	7	4	6
1	2	2	2	4	7
1	3	2	3	5	5
1	4	3	3	6	6
1	5	4	4	6	7
1	6	4	5	7	7

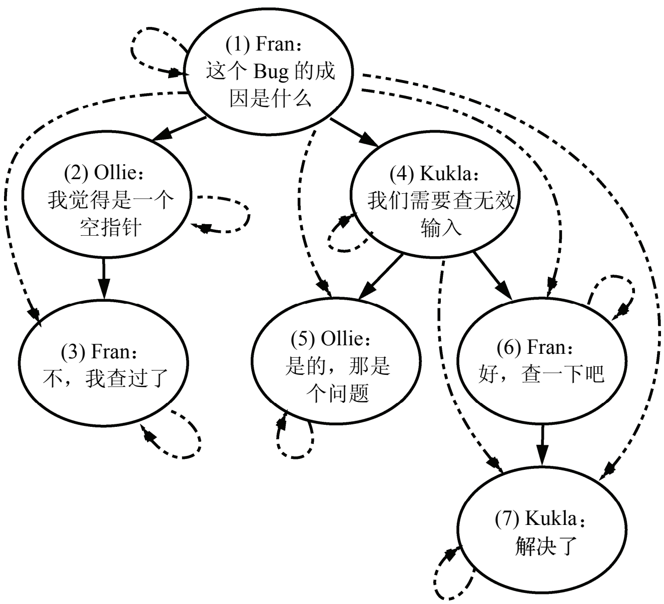


图 3-4 闭包表示意图

通过 TreePaths 表来获取祖先和后代比使用嵌套集更加地直接。例如要获取评论#4 的后代，

只需要在 TreePaths 表中搜索祖先是评论#4 的行就可以了：

Trees/soln/closure-table/descendants.sql

```
SELECT c.*
FROM Comments AS c
  JOIN TreePaths AS t ON c.comment_id = t.descendant
WHERE t.ancestor = 4;
```

要获取评论#6 的所有祖先，只需要在 TreePaths 表中搜索后代为评论#6 的行就可以了：

Trees/soln/closure-table/ancestors.sql

```
SELECT c.*
FROM Comments AS c
  JOIN TreePaths AS t ON c.comment_id = t.ancestor
WHERE t.descendant = 6;
```

要插入一个新的叶子节点，比如评论#5 的一个子节点，应首先插入一条自己到自己的关系，然后搜索 TreePaths 表中后代是评论#5 的节点，增加该节点和新插入节点的“祖先–后代”关系（包括评论#5 的自我引用）：

Trees/soln/closure-table/insert.sql

```
INSERT INTO TreePaths (ancestor, descendant)
  SELECT t.ancestor, 8
  FROM TreePaths AS t
  WHERE t.descendant = 5
 UNION ALL
  SELECT 8, 8;
```

要删除一个叶子节点，比如评论#7，应删除所有 TreePaths 表中后代为评论#7 的行：

Trees/soln/closure-table/delete-leaf.sql

```
DELETE FROM TreePaths WHERE descendant = 7;
```

要删除一棵完整的子树，比如评论#4 和它所有的后代，可删除所有在 TreePaths 表中后代为#4 的行，以及那些以评论#4 的后代为后代的行：

Trees/soln/closure-table/delete-subtree.sql

```
DELETE FROM TreePaths
WHERE descendant IN (SELECT descendant
                     FROM TreePaths
                     WHERE ancestor = 4);
```

请注意，如果你删除了 TreePahts 中的一条记录，并不是真正删除了这条评论。这对于评论系统这个例子来说可能很奇怪，但它在其他类型的树形结构的设计中会变得比较有意义。比如在产品目录的分类或者员工组织架构的图表中，当你改变了节点关系的时候，并不是真地想要删除一个节点。我们把关系路径存储在一个分开独立的表中，使得设计更加灵活。

要从一个地方移动一棵子树到另一地方，首先要断开这棵子树和它的祖先们的关系，所需要

做的就是找到这棵子树的顶点，删除它的所有子节点和它的所有祖先节点间的关系。比如将评论#6 从它现在的位置（评论#4 的孩子）移动到评论#3 下，首先做如下的删除（确保别把评论#6 的自我引用删掉）。

Trees/soln/closure-table/move-subtree.sql

```
DELETE FROM TreePaths
WHERE descendant IN (SELECT descendant
                     FROM TreePaths
                     WHERE ancestor = 6)
  AND ancestor IN (SELECT ancestor
                   FROM TreePaths
                   WHERE descendant = 6
                     AND ancestor != descendant);
```

查询评论#6 的祖先（不包含评论#6 自身），以及评论#6 的后代（包括评论#6 自身），然后删除它们之间的关系，这将正确地移除所有从评论#6 的祖先到评论#6 和它的后代之间的路径。换言之，这就删除了路径(1, 6)、(1,7)、(4, 6)和(4, 7)，并且它不会删除(6, 6) 或 (6, 7)。

然后将这棵孤立的树和新节点及它的祖先建立关系。可以使用 CROSS JOIN 语句来创建一个新节点及其祖先和这棵孤立的树中所有节点间的笛卡儿积来建立所有需要的关系。

Trees/soln/closure-table/move-subtree.sql

```
INSERT INTO TreePaths (ancestor, descendant)
  SELECT supertree.ancestor, subtree.descendant
  FROM TreePaths AS supertree
    CROSS JOIN TreePaths AS subtree
  WHERE supertree.descendant = 3
    AND subtree.ancestor = 6;
```

这样就创建了评论#3 及它的所有祖先节点到评论#6 及其所有后代之间的路径。因此，新的路径是：(1, 6)、(2, 6)、(3, 6)、(1, 7)、(2, 7)、(3, 7)。同时，评论#6 为顶点的这棵子树就成为了评论#3 的后代。笛卡儿积能创建所有需要的路径，即使这棵子树的层级在移动过程中发生了改变。

闭包表的设计比嵌套集更加地直接，两者都能快捷地查询给定节点的祖先和后代，但是闭包表能更加简单地维护分层信息。这两个设计都比使用邻接表或者路径枚举更方便地查询给定节点的直接后代和父代。

然而，你可以优化闭包表来使它更方便地查询直接父亲节点或子节点：在 TreePaths 表中增加一个 path_length 字段。一个节点的自我引用的 path_length 为 0，到它直接子节点的 path_length 为 1，再下一层为 2，以此类推。查询评论#4 的子节点就变得很直接：

Trees/soln/closure-table/child.sql

```
SELECT *
FROM TreePaths
WHERE ancestor = 4 AND path_length = 1;
```

3.5.4　你该使用哪种设计

每种设计都各有优劣，如何选择设计依赖于应用程序中的哪种操作最需要性能上的优化。在图 3-5 中，操作依据每种树的设计被标记为简单或者困难。你也可以参考以下列出的每种设计的优缺点。

设　计	表	查询子	查询树	插　入	删　除	引用完整性
邻接表	1	简单	困难	简单	简单	是
递归查询	1	简单	简单	简单	简单	是
枚举路径	1	简单	简单	简单	简单	否
嵌套集	1	困难	简单	困难	困难	否
闭包表	2	简单	简单	简单	简单	是

图 3-5　层级数据设计比较

- 邻接表是最方便的设计，并且很多软件开发者都了解它。
- 如果你使用的数据库支持 `WITH` 或者 `CONNECT BY PRIOR` 的递归查询，那能使得邻接表的查询更为高效。
- 枚举路径能够很直观地展示出祖先到后代之间的路径，但同时由于它不能确保引用完整性，使得这个设计非常地脆弱。枚举路径也使得数据的存储变得比较冗余。
- 嵌套集是一个聪明的解决方案——但可能过于聪明了，它不能确保引用完整性。最好在一个查询性能要求很高而对其他需求要求一般的场合来使用它。
- 闭包表是最通用的设计，并且本章所描述的设计中只有它能允许一个节点属于多棵树。它要求一张额外的表来存储关系，使用空间换时间的方案减少操作过程中由冗余的计算所造成的消耗。

关于存储和操作 SQL 中的分层数据有很多东西可以说。Jeo Celko 的 *Trees and Hierarchies in SQL for Smarties*[Cel04]是一本介绍分层查询的好书，另一本讲解了树及图论的书是 *SQL Design Patterns*[Tro06]，作者是 Vadim Tropashko。后者更加正规及理论化。

一个分层数据结构包含了数据项和它们之间的关系。

需要合理的设计两者的模型来配合你的工作。

第 4 章 需要 ID

外面的生物从猪看到人，从人看到猪，再从猪看到人，但它们已经分辨不出谁是猪谁是人了。

► 乔治 • 奥威尔，《动物庄园》

最近，有个程序员请教我一个很常见的问题：如何阻止表中出现重复项。 一开始，我认为可能是他的表中缺少一个主键，但后来发现并不是这么回事。

他的内容管理数据库中存储了在一个网站上所发表的文章。他使用一个交叉表来存储文章和标签这两张表之间的多对多的关系。

ID-Required/intro/articletags.sql

```
CREATE TABLE ArticleTags (
  id          SERIAL PRIMARY KEY,
  article_id  BIGINT UNSIGNED NOT NULL,
  tag_id      BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (article_id) REFERENCES Articles (id),
  FOREIGN KEY (tag_id)     REFERENCES Tags (id)
);
```

在他尝试通过给定的标签来查询文章数量的时候，返回了错误的结果。他知道“economy”标签只有 5 篇文章，但是查询结果告诉他有 7 篇。

ID-Required/intro/articletags.sql

```
SELECT tag_id, COUNT(*) AS articles_per_tag FROM ArticleTags WHERE tag_id = 327;
```

使用 327 这个 tag_id 去查询时，他看到这个标签和同一篇文章关联了三次，三条记录显示了同样的关系，尽管三条记录显示的关系 id 不同。

id	tag_id	article_id
22	327	1234
23	327	1234
24	327	1234

这张表是有主键的，但是主键并没有办法阻止像上表中那样的重复。有一种解决方案是对另外两列创建一个 UNIQUE 约束，但同时就会使得 id 这一列显得非常多余：为什么还需要它呢？

4.1 目标：建立主键规范

这章的目标就是要确认那些使用了主键，却混淆了主键的本质而造成的一种反模式。

每个了解数据库设计的人都知道，主键对于一张表来说是一个很重要，甚至必需的部分。这确实是事实，主键是好的数据库设计的一部分。主键是数据库确保数据行在整张表中唯一性的保障，它是定位到一条记录并且确保不会重复存储的逻辑机制。主键也同时可以被外键引用来建立表与表之间的关系。

难点是选择哪一列作为主键。大多数表中的每个属性的值都有可能被很多行使用。例如一个人的姓和名就一定会在表中重复出现，即使电子邮件地址或者美国社保编号或者税单编号也不能保证绝对不会重复。

在这样的表中，需要引入一个对于表的域模型无意义的新列来存储一个伪值。这一列被用作这张表的主键，从而通过它来确定表中的一条记录，即便其他的列允许出现适当的重复项。这种类型的主键列我们通常称其为伪主键或者代理键。

大多数的数据库提供一种和当前处理事务无关的底层方案，来确保每次都能生成全局唯一的一个整数作为伪主键，即使客户端此时正发起并发操作。

真的需要一个主键吗？

我曾经听一些开发人员声称他们的数据库表不需要主键。

有时候这些程序员想要避免想象中的维护唯一索引的开销，或者他们的表中并没有实现这一目的的列。

当你需要做下面这些事情的时候，主键约束是很重要的：

- 确保一张表中的数据不会出现重复行；
- 在查询中引用单独的一行记录；
- 支持外键。

如果你不使用主键约束，就只有一个选择：检查是否有重复行。

```
SELECT bug_id FROM Bugs GROUP BY bug_id HAVING COUNT(*) > 1;
```

多久需要执行一次这样的检查？当你找到了一条重复记录时，又要如何处理？

一张没有主键的表就好像你的 MP3 播放列表里没有歌名一样。你依然可以听歌，但无法找到想听的那首歌，也没办法确保播放列表中没有重复的歌曲。

伪主键直到 SQL:2003 才成为一个标准，因而每个数据库都使用自己特有的 SQL 扩展来实现伪主键，甚至不同数据库中对于伪主键都有不同的名称（不同的表述），如下表。

特　　性	支持的数据库
AUTO_INCREMENT	MySQL
GENERATOR	Firebird, InterBase
IDENTITY	DB2, Derby, Microsoft SQL Server, Sybase
ROWID	SQLite
SEQUENCE	DB2, Firebird, Informix, Ingres, Oracle, PostgreSQL
SERIAL	MySQL, PostgreSQL

伪主键是非常有用的数据库特性，但并不是声明主键的唯一解决方案。

4.2　反模式：以不变应万变

很多的书、文章以及程序框架都会告诉你，每个数据库的表都需要一个主键，且具有如下三个特性：

- 主键的列名叫做 id；
- 数据类型是 32 位或者 64 位整型；
- 主键的值是自动生成来确保唯一的。

在每张表中都存在一个叫做 id 的列是如此地平常，甚至 id 已经成为了主键的同义词。很多程序员在一开始学习 SQL 时就被灌输了错误的概念，认为主键就是像如下的程序那样定义的一列。

ID-Required/anti/id-ubiquitous.sql

```
CREATE TABLE Bugs (
  id          SERIAL PRIMARY KEY,
  description VARCHAR(1000),
  -- . . .
);
```

给每张表都增加一列 id，使其使用显得太过随意。

4.2.1　冗余键值

你可能会发现在一张表中定义了 id 这一列作为主键，仅仅因为这么做符合传统，然而可能又同时存在另一列从逻辑上来说更为自然的主键，这一列甚至也具有 UNIQUE 约束。比如，在 Bugs 这张表中，程序会使用这个 Bug 所属项目的助记符或者其他的标识信息来标记一个 Bug。

ID-Required/anti/id-redundant.sql

```
CREATE TABLE Bugs (
```

```
  id          SERIAL PRIMARY KEY,
  bug_id      VARCHAR(10) UNIQUE,
  description VARCHAR(1000),
  -- . . .
);

INSERT INTO Bugs (bug_id, description, ...)
  VALUES ('VIS-078', 'crashes on save', ...);
```

bug_id 这一列和 id 有着相似的功能，都是为了唯一地标识一条记录。

4.2.2 允许重复项

一个组合键包含了多个不同的列。组合键的典型场景是在像 BugsProducts 这样的交叉表中。主键需要确保一个给定的 bug_id 和 product_id 的组合在整张表中只能出现一次，虽然同一个值可能在很多不同的配对中出现。

然而，当你使用 id 这一列作为主键，约束就不再是 bug_id 和 product_id 的组合必须唯一了。

ID-Required/anti/superfluous.sql

```
CREATE TABLE BugsProducts (
  id          SERIAL PRIMARY KEY,
  bug_id      BIGINT UNSIGNED NOT NULL,
  product_id  BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1), (1234, 1), (1234, 1); -- 重复项也是可以输入的
```

当你用这张交叉表去查询 Bugs 和 Products 的关系时，重复项会引起意料之外的结果。要确保没有重复项，你可以在 id 之外，额外声明另外两列需要一个 UNIQUE 约束。

ID-Required/anti/superfluous.sql

```
CREATE TABLE BugsProducts (
  id          SERIAL PRIMARY KEY,
  bug_id      BIGINT UNSIGNED NOT NULL,
  product_id  BIGINT UNSIGNED NOT NULL,
  UNIQUE KEY (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

但是，当你在 bug_id 和 product_id 这两列上应用了唯一性约束，id 这一列就会变成多余的。

4.2.3　意义不明的关键字

单词“code”有很多意思，其中之一便是在交流中用于简化或者加密消息的。“code”还有一个意思就是写代码。在程序设计中，我们需要做的是尽量使得逻辑更加地清晰明了，而不是“使其复杂到难以辨认”。

“id”这个词是如此地普通，完全无法表达更深层次的意思，特别是在你做两张表的联结查询，而它们都有一个叫做 id 的主键时。

ID-Required/anti/ambiguous.sql

```
SELECT b.id, a.id
FROM Bugs b
JOIN Accounts a ON (b.assigned_to = a.id)
WHERE b.status = 'OPEN';
```

如果你的程序用的是列名而不是列在表中的序号来编码，该如何区分缺陷 id 和账号 id？在像 PHP 一样的动态语言中，这个问题显得更加突出。比如你需要获得一个关联数组的查询结果，除非在查询时指定了列别名，否则其中的一个 id 列会覆盖掉另一列 id 的值。

列名 id 并不会使查询变得更加清晰。但如果列名叫做 bug_id 或者 account_id，事情就会变得更加简单。我们使用主键来唯一地定位一条记录，因此主键的列名就应该更加便于理解。

4.2.4　使用USING关键字

可能你很熟悉联结查询的语法，使用 SQL 关键字 JOIN 和 ON 来处理两张表的匹配数据行，就像下面的例子：

ID-Required/anti/join.sql

```
SELECT * FROM Bugs AS b JOIN BugsProducts AS bp ON (b.bug_id = bp.bug_id);
```

SQL 同时也支持另一种更加简洁的表达式来表示两张表的联结。如果两张表都有同样的列名，就可以用如下的表达式来重写上面的需求：

ID-Required/anti/join.sql

```
SELECT * FROM Bugs JOIN BugsProducts USING (bug_id);
```

然而，如果所有的表都要求定义一个叫做 id 的伪主键，那么作为外键的列将永远不能使用和引用的列相同的列名。同时，每次查询都必须使用啰嗦的 ON 表达式：

ID-Required/anti/join.sql

```
SELECT * FROM Bugs AS b JOIN BugsProducts AS bp ON (b.id = bp.bug_id);
```

4.2.5 使用组合键之难

一些开发人员拒绝使用组合键，因为他们觉得它太难以使用了。如果要比较两个键值，必须比较其包含的所有列的值；一个引用组合键的外键，其本身也必须是一个组合外键。此外，使用组合键需要打更多的字。

程序员拒绝在设计中使用组合键，就好像数学家拒绝使用二维或者三维坐标系，只使用一维数轴来计算所有现实事物一样。虽然那样的确会使得几何学变得简单，却无法描绘我们的真实世界。

特定范围序列

有些程序员通过给当前使用的最大值加 1 来获取一条新记录的 ID：

```
SELECT MAX(bug_id) + 1 AS next_bug_id FROM Bugs;
```

当客户端发起并发请求来获取新记录的 id 时，这样的做法并不可靠。同样的值可能被多个客户端同时获得，这样的情况称为“竞争”。

要避免由多客户端引起的竞争问题，就需要在计算并重新设定最大值时阻止并发的插入，你不得不锁住整张表——单纯锁住行并不够。锁住整张表的访问会造成系统性能的瓶颈，因为它使得所有的并发访问必须排队。

序列通过将运算和事务在逻辑上分离来解决并发问题。序列确保即使在多并发下，每次调用都会返回不同的值，因而无论是否需要将由序列返回的值插入新行，序列都不会再次生成同样的值。由于序列的这种特性，多个客户端可以同时发起请求，并且确信它们获取到的值在每个客户端中是唯一的。

大多数数据库都支持一些内置函数来获取一个序列生成的最后一个值。比如，MySQL 中的这个函数叫做 LAST_INSERT_ID()；Microsoft SQL Server 使用叫做 SCOPE_IDENTITY() 的函数，Oracle 中称为 *SequenceName*.CURRVAL()。

这些函数都返回在自己的会话周期内生成的值，即使有其他的客户端同时在调用也是如此，因而不会产生竞争。

4.3 如何识别反模式

这章的反模式的特征很容易辨认：使用了过于普通的 id 作为表的主键的列名。实际上，绝无理由不使用另一个更加有意义的名称。

如果你遇到了下面的几个问题，可能是使用了这个反模式的征兆。

- “我觉得这张表不需要主键。”
  会这么说的开发人员一定是误解了“主键”和“伪主键”的含义。每张表都必须有一个

主键来确保不出现重复项并定位每一行。他们可能想要一个更自然的列名来做主键或者需要一个组合键。

- “我怎么能在多对多的表中存储重复的项？”
  在一个多对多关系的交叉表中需要声明一个主键约束，或者至少需要有一个针对那些被引用为外键的列的唯一约束。
- “我学过数据库设计理论，里面说我应该把数据移到一张查询表中，然后通过 ID 查找。但是我不想这么做，因为每次我想要获得真实的数据，都不得不做一次联结查询。”
  这在数据库设计中是一个常见的误区，称为“正规化”（normalization），然而实际中对于伪主键并没有什么需要做的。更详细的信息参考附录 A。

4.4 合理使用反模式

一些面向对象的框架假设“惯例优于配置”从而简化其设计。它们期望每张表都使用同样的方法来定义它的主键：使用 `id` 作为列名，并且使用类型为整型的伪主键。如果你使用了这样的一个框架，就可能不得不遵守这样的约定，才能够进一步使用这个框架所提供的其他特性。

使用伪主键，或者通过自动增长的整型的机制本身没有什么错误，但不是每张表都需要一个伪主键，更没有必要将每个伪主键都定义成 `id`。

对于太长而不方便实现的自然键来说，伪主键是很好的代替品。比如在一个记录文件系统中所有文件属性的表中，文件路径是一个很好的自然键，但对一个字符串列做索引的开销会很大。

4.5 解决方案：裁剪设计

主键是约束而非数据类型。你可以定义任意列或任意多的列为主键，只要其数据类型支持索引。同时，还可以将一个列的数据类型定义为自增长的整型而不设定其为主键。这两者是完全无关的。

别被既有的惯例限制住设计。

4.5.1 直截了当地描述设计

为主键选择更有意义的名称：一个能够反应这个主键所代表的实体的类型的名字。比如，`Bugs` 这张表的主键应该叫做 `bug_id`。

外键应该尽可能地和所引用的列使用相同的名称，这通常意味着：一个主键的名称应该在整个数据库的设计中唯一；任意两张表都不应该使用相同的名称来定义主键，除非其中之一引用了

另一个作为外键。然而，凡事都有例外，有时外键的名称需要和其所引用的主键区分开，从而使得它们之间的引用关系表现得更加清晰。

ID-Required/soln/foreignkey-name.sql

```
CREATE TABLE Bugs (
  -- . . .
  reported_by  BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
);
```

信息行业中存在一个叫做 ISO/IEC 11179[①]的现行标准，用来描述元数据的命名惯例。换言之，这份标准就是用来指导你如何更合理地给数据库表中的每一列命名。就像大多数的 ISO 标准一样，这份标准文档也几乎是一份天书，但 Joe Celko 在他的 *SQL Programming Stytle*[Cel05]一书中实践了这份标准。

4.5.2 打破传统

面向对象的框架希望你使用 `id` 这个伪主键，但同时也允许无视这个规则转而使用别的名字。下面是 Ruby on Rails 的一个例子：

ID-Required/soln/custom-primarykey.rb

```
class Bug < ActiveRecord::Base
  set_primary_key "bug_id"
end
```

一些开发人员认为仅仅在处理那些遗留数据库而无法使用自己所喜欢的规范时，才需要为主键列定义不同的名称，而事实上，即使对于新项目，为每一列指定一个有意义的名称也十分重要。

4.5.3 拥抱自然键和组合键

如果你的表中包含一列能确保唯一、非空以及能够用来定位一条记录，就别仅仅因为传统而觉得有必要再加上一个伪主键。

实践证明，一张表中的每一列都在最初的设计之后遭遇改变或者一开始就是不唯一的，这是再平常不过的事情。数据库的设计趋向于在整个项目的生命周期中不断地调整和优化，并且决策者也可能一点也不在乎自然键的“神圣不可侵犯”。有时侯，一个列在最开始时像是个很好的自然键，但随后又允许合法的重复项。此时，伪主键便成了唯一的选择。

在合适的时候也可以使用组合键，比如一条记录可以通过多列的组合完全定位，就像 `BugsProducts` 表，那就通过那些列创建一个组合键吧。

① http://metadata-standards.org/11179/。

ID-Required/soln/compound.sql

```
CREATE TABLE BugsProducts (
  bug_id      BIGINT UNSIGNED NOT NULL,
  product_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1), (1234, 2), (1234, 3);
INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1); -- 错误：重复项
```

需要注意的是，一个引用了组合键的外键同样需要是一个组合键。这看上去像是很累赘地对其依赖的列做了一个副本，但这么做也有好处，它能简化原来需要一个联结查询才能获得的一条记录的相关属性。

规范仅仅在它有帮助时才是好的。

第 5 章 不用钥匙的入口

是故胜兵先胜而后求战，败兵先战而后求胜。

► 孙子

“比尔，我们实验室中的同一台服务器好像在同一天被两个经理预定了——这怎么可能？”测试实验室的经理冲进了我的小隔间说道：“你能查一下是怎么回事，然后解决一下这个问题吗？他们都对着我大吼大叫，说他们需要这台设备，并且说我耽误了他们的项目进度。”

几年前我使用 MySQL 设计了一个设备跟踪系统。MySQL 默认的存储引擎是 MyISAM，一个并不支持外键约束的东西。数据库的设计中有很多的逻辑关系，但无法保证引用完整性。

当程序不断地更新，并且使用了新的方法来操作数据的时候，我们制造了一个问题：当无法保障引用完整性时，报表中出现了不一致的情况，各部分的计算结果不一致，最终导致了同时预约的结果。

项目经理让我写一个监控脚本，然后让它在后台持续地执行，检查是否有不一致的情况发生，比如数据库中是否存在孤立的记录，并且当有此类错误发生时，通过电子邮件报警通知我们。

数据库中每个表与表的关系都需要使用这些脚本来进行检查。随着数据量和表的数目不断增长，监控脚本的查询量和相应的查询时间也随之增长，报警的邮件也变得很长。这样的场景，你是否似曾相识？

监控脚本工作得很好，但这显然是个很浪费的重复造轮子[①]工程。我所需要的是能找到一个方法，使得程序在用户提交了错误数据时能及时地发现。外键约束能做到吗？

5.1 目标：简化数据库架构

关系数据库的设计基本上可以说就是关于每张独立表之间的关系的设计。引用完整性是合理

① reinvent the wheel：表示重复发明或开发已经存在的东西。——编者注

的数据库设计和操作的非常重要的一部分。当一列或者多列声明了外键约束后，这些列中的数据必须在其父表（即所引用的表）的主键列或者唯一字段的列中存在。原理貌似很简单。

然而，一些开发人员不推荐使用引用完整性约束。你可能听说过这么几点不使用外键的原因。

- 数据更新有可能和约束冲突。
- 当前的数据库设计如此灵活，以致于不支持引用完整性约束。
- 数据库为外键建立的索引会影响性能。
- 当前使用的数据库不支持外键。
- 定义外键的语法并不简单，还需要查阅。

5.2　反模式：无视约束

即使第一感觉告诉你，省略外键约束能使得数据库设计更加简单、灵活，或者执行更加高效，你还是不得不在其他方面付出相应的代价——必须增加额外的代码来手动维护引用完整性。

5.2.1　假设无瑕代码

很多人对引用完整性的解决方案是通过编写特定的程序代码来确保数据间的关系的。每次插入新记录时，需要确保外键列所引用的值在其对应的表中存在；每次删除记录时，需要确保所有相关的表都要同时合理地更新。用时下流行的话来说就是：千万别犯错（make no mistakes）。

要避免在没有外键约束的情况下产生引用的不完整状态，需要在任何改变生效前执行额外的 SELECT 查询，以此来确保这些改变不会导致引用错误。比如，在插入一条新记录之前，需要检查对应的被引用记录是否存在：

Keyless-Entry/anti/insert.sql

```
SELECT account_id FROM Accounts WHERE account_id = 1;
```

然后才可以添加一个引用了这个账号的 bug 记录：

Keyless-Entry/anti/insert.sql

```
INSERT INTO Bugs (reported_by) VALUES (1);
```

要删除一条记录，需要先确认没有别的记录引用了该条记录：

Keyless-Entry/anti/delete.sql

```
SELECT bug_id FROM Bugs WHERE reported_by = 1;
```

随后才能删除这个账号：

Keyless-Entry/anti/delete.sql

```
DELETE FROM Accounts WHERE account_id = 1;
```

如果这个 account_id 为 1 的用户，恰巧在删除操作的查询和删除语句的执行间隙插入了一条新的缺陷记录，该怎么办？这看上去不太可能发生，但是戈登·莱顿（DOS 4 的架构师）曾经说过一句很著名的话："不怕一万，就怕万一。"这样的确会造成一个破碎的引用关系——一个 bug 由一个不存在的账号提交。

唯一的做法是在检查数据时显式地锁住 Bugs 这张表，然后在删除账号完成之后再解锁。任何需要这样加锁的架构，在高并发和大数据量查询时的表现都非常糟糕。

5.2.2 检查错误

本章所描述的不正确的解决方案使用的是程序员写的外部脚本来检查错误的数据。

举例来说，在我们的缺陷数据库中，Bugs.status 一列引用了 BugStatus 这张表。为了找到状态值有异常的缺陷记录，你可能会使用如下的查询语句：

Keyless-Entry/anti/find-orphans.sql

```
SELECT b.bug_id, b.status
FROM Bugs b LEFT OUTER JOIN BugStatus s
  ON (b.status = s.status)
WHERE s.status IS NULL;
```

可以想象一下，你需要为数据库中所有的引用关系写类似的脚本。

如果你发现自己正陷于使用这样的方法检查数据库中错误的引用关系的窘境时，下一个问题便是，多久需要执行一次这个脚本？每天手动执行成百上千或者更多次这样的查询脚本，是一件非常乏味的事情。

当你真的发现了一个错误的引用关系时，该怎么办？你能修复它吗？有时候也许行。比如，可以将一个无意义的值改成默认值[1]。

Keyless-Entry/anti/set-default.sql

```
UPDATE Bugs SET status = DEFAULT WHERE status = 'BANANA';
```

不可避免，还有很多情况是无法通过简单设为默认值就能处理的。比如，Bugs.reported_by 这一列应该要引用一个提交这个缺陷的账号，而当这个值是无效值时，应该使用哪个账号来代替？

5.2.3 "那不是我的错！"

所有与数据库相关的代码都是完美的——这基本上是不可能的。你可以简单地使用几个函数来处理相似的数据更新，但是如果需要修改代码的时候，要怎么保证应用程序中的每一个相关点

① SQL 支持使用 DEFAULT 关键字。

都一起修改了呢？

而且可能有的用户直接操作了数据库，用了 SQL 查询工具或者使用了私有脚本。通过使用临时编写的 SQL 语句，很容易产生错误的引用。你应该假设这些事情会在你的应用程序生命周期的某一个时刻会发生。

你需要数据库中的数据保持连贯，这意味着，需要仰仗数据库中的引用关系始终是正常的，不出错的。但你不能确定所有的应用程序或者脚本在访问数据库时所作的事情都是正确合理的。

5.2.4 进退维谷

很多开发人员避免使用外键约束的理由，是因为这些约束会使得更新多张表中相关联的列变得比较麻烦。比如，如果你想要删除一条被其他记录所依赖的记录，就不得不删除所有的子记录来避免违反外键约束：

Keyless-Entry/anti/delete-child.sql

```sql
DELETE FROM BugStatus WHERE status = 'BOGUS'; -- ERROR!

DELETE FROM Bugs WHERE status = 'BOGUS';

DELETE FROM BugStatus WHERE status = 'BOGUS'; -- retry succeeds
```

你不得不为每张子表手动执行多条语句。如果你在将来的需求变更下又往数据库中添加了一张新表，就不得不修改所有相关的代码来删除这张新表中的数据。但这个问题是可以解决的。

而没解决的问题是，当你 UPDATE 一条被其他记录依赖的记录时，在没有更新父记录前，你不能更新子记录，而且也不能在更新父记录前更新子记录。你需要同步执行两边的更新，但是使用两个独立的更新语句是不现实的。这就是所谓的进退维谷。

Keyless-Entry/anti/update-catch22.sql

```sql
UPDATE BugStatus SET status = 'INVALID' WHERE status = 'BOGUS'; -- ERROR!

UPDATE Bugs SET status = 'INVALID' WHERE status = 'BOGUS'; -- ERROR!
```

一些开发人员发现这样的情况几乎无法管理，因而他们决定干脆不使用外键。但是，我们稍后将会看到外键如何用一种简单而高效的方法同时更新和删除多张表中的记录。

5.3 如何识别反模式

如果你听到有人说的话像下面列举的这样，他们可能正使用了本章所描述的反模式。

- “我要怎么写这个查询语句来检查一个值是否没有同时在两张表中存在？”
  通常这样的需求是为了查找那些孤立的行。

- "有没有一种简单的方法来判断在一张表中存在的数据是否也在第二张表中存在？"
  这么做是用来确认父记录切实存在。外键会自动完成这些，并且外键会使用父表的索引尽可能高效地完成。
- "外键？有人告诉我别用它，因为那会影响数据库的效率。"
  性能总是用来裁剪设计的一个很好的理由，但总是会引入更多的问题，甚至包括性能问题本身。

5.4 合理使用反模式

有时你被迫使用不支持外键约束的数据库产品（比如 MySQL 的 MyISAM 存储引擎，或者比 SQLite 3.6.19 早的版本）。如果是这种情况，那你不得不使用别的方法来弥补，比如说前文描述的监控脚本。

同样也存在一些极度灵活的数据库设计，外键无法用来表示其对应的关系。如果你不能使用传统的引用完整性约束，很有可能你正在使用另一个 SQL 反模式。可以阅读第 6 章和第 7 章来获取更多的信息。

5.5 解决方案：声明约束

日语中有个短语 poka-yoke，意思是"防差错技术"。这是一种制造工艺，通过在错误发生时对错误加以阻止、纠正或者引起注意来帮助消除产品缺陷。这项工艺能够显著地提升产品质量，帮助减少纠错的必要，相比于使用这种工艺的开销，其所获得收益更高。

你可以将"防差错技术"的理论应用到你的数据库设计中——通过使用外键来确保引用完整性。相对于查找并修正完整性错误，你可以在进入数据库的第一道关卡上就阻止这样的错误发生。

Keyless-Entry/soln/foreign-keys.sql

```
CREATE TABLE Bugs (
  -- . . .
  reported_by       BIGINT UNSIGNED NOT NULL,
  status            VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);
```

那些现存的代码以及临时的查询都将遵守同样的约束，因而，不会给任何被遗忘的代码片段或者其他的访问方式绕开约束的方法。数据库本身就会拒绝所有不合理的改变，无论这个改变是通过什么方式造成的。

通过使用外键，能够避免编写不必要的代码，同时还能确保一旦修改了数据库中的内容，所有的代码依旧能够用同样的方式执行。这节省了大量开发、调试以及维护时间。软件行业中每千行代码的平均缺陷数约为 15~50 个。在其他条件相同的情况下，越少的代码，意味着越少的缺陷。

5.5.1　支持同步修改

外键有另一个在应用程序中无法模拟的特性：级联更新。

Keyless-Entry/soln/cascade.sql

```
CREATE TABLE Bugs (
  -- . . .
  reported_by       BIGINT UNSIGNED NOT NULL,
  status            VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
    ON UPDATE CASCADE
    ON DELETE RESTRICT,
  FOREIGN KEY (status) REFERENCES BugStatus(status)
    ON UPDATE CASCADE
    ON DELETE SET DEFAULT
);
```

这个解决方案允许你更新或者删除父记录，并且让数据库来处理那些引用了父记录的子记录。更新父表 BugStatus 和 Accounts 会自动地应用到 Bugs 表中的子记录。这就不会造成进退维谷的状况了。

在外键约束中声明 ON UPDATE 和 ON DELETE 的方式允许你控制级连操作的结果。比如说，在 reported_by 这个外键上声明的 RESTRICT 意味着你无法删除一个在 Bugs 这张表中被引用的账号。这个约束会阻止删除操作并且产生一个错误。而无论什么情况下，你删除一个 status 值，任何引用该值的记录都将被设置成默认值。

在执行更新和删除两个操作中的任意一个时，数据库都会自动修改两张表中的数据。外键的引用状态在操作之前和之后都将保持完好。

如果你往数据库中新加入一张子表，子表中的外键就规定了级联操作的行为。你不需要修改任何应用程序的代码。同时，无论多少张子表引用了同一张父表，都不需要改变任何东西。

5.5.2　系统开销过度？不见得

的确，外键约束需要多那么一点额外的系统开销，但相比于其他的一些选择，外键确实更高效一点。

- 不需要在更新或删除记录前执行 SELECT 进行检查。
- 在同步修改时不需要再锁住整张表。
- 不再需要执行定期的监控脚本来修正不可避免的孤立数据。

外键使用方便，提高性能，还能帮助你在任何简单或复杂形式的数据变更下始终维持引用完整性。

通过使用约束来帮助数据库防止错误。

第 6 章 实体-属性-值

> 如果你想把一只猫肢解了来研究它是怎么工作的，那么首先你要得到一只不工作的猫。
>
> ▶ 道格拉斯·亚当斯

“我要怎么按日期来统计记录条数？”对于数据库程序员来说，这是一个最基本的任务例子。它的解决方案在任何 SQL 入门介绍中都会出现，因为它包含了 SQL 最基本的语法：

EAV/intro/count.sql

```
SELECT date_reported, COUNT(*)
FROM Bugs
GROUP BY date_reported;
```

然而，这个简单的解决方案需要两个假设。

- 所有的值都存在同一列中，比如 `Bugs.data_reported`。
- 数据类型是可以比较的，因而 GROUP BY 可以通过比较两个值是否相等来分组。

假如这些假设不能满足呢？如果日期存储在 `date_reported` 或者 `report_date`，或者其他任何列且每条记录的列名都不相同呢？如果日期的格式各式各样，而数据库无法简单地比较两个日期又该如何呢？

如果你在使用“实体-属性-值”反模式，就会遇到前面说的以及其他的一些问题。

6.1 目标：支持可变的属性

可扩展性是所有软件项目设计中最普遍的一个目标。我们都想设计出一个不需要过多修改，甚至不需要修改就能适合将来需求变更的软件。

这并不是一个新的课题，自 1970 年 E. F. Codd 在他的 *A Relational Model of Data for Large Shared Data Banks*[Cod 70]一文中第一次介绍了关系模型的概念以来，相似的关于关系数据模型不灵活

性的争论就一直在持续。

通常来说，一张表由一些属性列组成，表中的每一条记录都使用这些列，因为每条记录表示的都是相似的对象实例。不同的属性集合表示不同的对象，因而就应该用不同的表来区分。

但是，在现代面向对象的编程模型中，不同的对象类型可能是相连的。比如，多个对象都可能是从同一个基类派生而来，它们既是实际子类的实例，也同时是父类的实例。我们可能想仅使用一张表来存储所有这些不同类型的对象，这样能方便进行比较和计算。但我们也需要将不同的子类分开存储，因为每个子类都有一些特殊的属性，和其他的子类甚至父类都不能共用。

我们继续使用 Bugs 数据库来举例。在图 6-1 中，`Bug` 和 `FeatureRequest` 有一些公共属性，我们将其提炼为一个基类，称为 `Issue`。每个事件都和一个报告它的人相关，同时也和一个产品相关，并且这个产品有个优先级用以比较。然而，`Bug` 有一些独特的属性：产生错误的产品版本号和错误的级别。同样地，`FeatureRequest` 也有自己的特有属性，比如，假设一个产品特性是和支持这一特性的开发赞助商相关的。

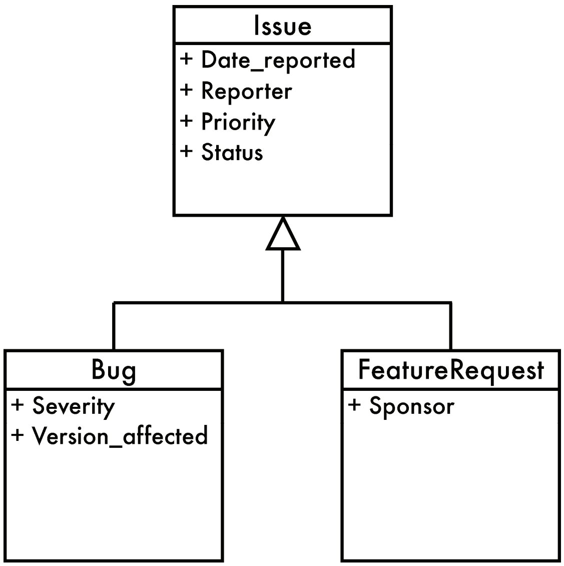


图 6-1　Bug 类型的类图

6.2　反模式：使用泛型属性表

对于某些程序员来说，当他们需要支持可变属性时，第一反应便是创建另一张表，将属性当成行来存储。图 6-2 中，属性表中的每条记录都包含三列。

- 实体：通常来说这就是一个指向父表的外键，父表的每条记录表示一个实体对象。

- 属性：在传统的表中，属性即每一列的名字，但在这个新的设计中，我们需要根据不同的记录来解析其标识的对象属性。
- 值：对于每个实体的每一个不同属性，都有一个对应的值。

  比如，一个给定的 Bug 是一个实体对象，我们通过它的主键来标识它，它的主键值为 1234。这个对象有一个属性 status，Bug 1234 的 status 的值为 NEW。

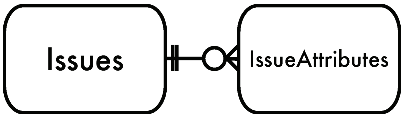


图 6-2 EAV 实体关系

这样的设计称为实体–属性–值，简称 EAV。有时也称之为：开放架构、无模式或者名–值对。

EAV/anti/create-eav-table.sql

```
CREATE TABLE Issues (
  issue_id    SERIAL PRIMARY KEY
);

INSERT INTO Issues (issue_id) VALUES (1234);

CREATE TABLE IssueAttributes (
  issue_id    BIGINT UNSIGNED NOT NULL,
  attr_name   VARCHAR(100) NOT NULL,
  attr_value  VARCHAR(100),
  PRIMARY KEY (issue_id, attr_name),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);
INSERT INTO IssueAttributes (issue_id, attr_name, attr_value)
  VALUES
    (1234, 'product',          '1'),
    (1234, 'date_reported',    '2009-06-01'),
    (1234, 'status',           'NEW'),
    (1234, 'description',      'Saving does not work'),
    (1234, 'reported_by',      'Bill'),
    (1234, 'version_affected', '1.0'),
    (1234, 'severity',         'loss of functionality'),
    (1234, 'priority',         'high');
```

通过增加一张额外的表，可以获得如下这些好处。

- 这两张表的列都很少。
- 新增的属性不会对现有的表结构造成影响，不需要新增列。
- 避免了由于空值而造成的表内容混乱。

这看上去是一个改良过的设计，然而，设计上的简单化并不足以弥补其造成的使用上的难度。

6.2.1 查询属性

假设你的领导想要每天获取一份 Bug 的报表，在传统的表结构设计中，Issues 表中仅包含了很简单的属性列，比如 date_reported，要根据日期查询 Bug 记录，只需要执行一句很简单的语句：

EAV/anti/query-plain.sql

```
SELECT issue_id, date_reported FROM Issues;
```

要使用 EAV 设计来做相同的事情，就需要先从表 IssueAttributes 中提取出属性列为 date_reported 的所有记录。查询操作更加啰嗦，而且不够清晰：

EAV/anti/query-eav.sql

```
SELECT issue_id, attr_value AS "date_reported"
FROM IssueAttributes
WHERE attr_name = 'date_reported';
```

6.2.2 支持数据完整性

使用了 EAV 的设计，需要放弃很多传统的数据库设计所带来的方便之处。

6.2.3 无法声明强制属性

要让你的领导能够顺利地生成项目报表，需要确保 date_reported 这个属性有值。在传统的数据库设计中，可以很简单地通过在声明的时候加上 NOT NULL 的限制来确保该列的值不为空。

在 EAV 的设计中，每个属性对应 IssueAttributes 表中的一行，而不是一列。你可能需要一个约束来检查对于每个 issue_id 都存在这么一行，并且这行的 attr_name 列的值是 date_reported。

然而，SQL 没有任何类型的约束支持这么做。因而你必须通过编写外部程序的代码确保这点。如果找到一个没有提交日期的 Bug 记录，应该为它加上一个日期吗？那应该给它赋什么值？如果胡乱猜测一个值或者使用默认值，对于你老板报表的精确性来说有多少影响？

6.2.4 无法使用SQL的数据类型

你的老板告诉你他的报表有点问题，因为人们输入的日期格式各式各样，甚至有时是个字符串而根本就不是一个日期。在传统的数据库中，你可以通过将一列的类型声明为 DATE 来确保这种情况不会发生。

EAV/anti/insert-plain.sql

```
INSERT INTO Issues (date_reported) VALUES ('banana'); -- ERROR!
```

在 EAV 的设计中，IssueAttributes.attri_value 列的数据类型就是一个单纯的字符串，从而才能仅用一列来适应任何可能的数据类型。因此，没有好办法来阻止无效数据的录入。

EAV/anti/insert-eav.sql

```
INSERT INTO IssueAttributes (issue_id, attr_name, attr_value)
  VALUES (1234, 'date_reported', 'banana');  -- Not an error!
```

有些人尝试扩展 EAV 的设计，为每一个 SQL 类型定义一个单独的 attr_value 列，不需要使用的列就留空。这可以让你使用 SQL 的数据类型，却使得查询变得更加恐怖：

EAV/anti/data-types.sql

```
SELECT issue_id, COALESCE(attr_value_date, attr_value_datetime,
  attr_value_integer, attr_value_numeric, attr_value_float,
  attr_value_string, attr_value_text) AS "date_reported"
FROM IssueAttributes
WHERE attr_name = 'date_reported';
```

你可能需要添加更多的列来支持用户自定义的数据类型或者域（domain）。

6.2.5 无法确保引用完整性

在传统的数据库中，你可以定义一个指向另一张表的外键来约束某些属性的取值范围。比如，一个 Bug 或者事件的 status 属性应该是一张很小的 BugStatus 表中的一个值。

EAV/anti/foreign-key-plain.sql

```
CREATE TABLE Issues (
  issue_id         SERIAL PRIMARY KEY,
  -- other columns
  status           VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);
```

在 EAV 的设计中，你无法在 attr_value 列上使用这种约束方法。引用完整性的约束会应用到表中的每一行。

EAV/anti/foreign-key-eav.sql

```
CREATE TABLE IssueAttributes (
  issue_id         BIGINT UNSIGNED NOT NULL,
  attr_name        VARCHAR(100) NOT NULL,
  attr_value       VARCHAR(100),
  FOREIGN KEY (attr_value) REFERENCES BugStatus(status)
);
```

如果你像这样定义约束，那会强制表中每个属性的值都必须存在于 BugStatus 中，而不仅仅是 status 属性。

6.2.6 无法配置属性名

你老板的报表依旧非常不靠谱。你发现这些属性的命名不够清晰。有个 Bug 的属性叫做 date_reported，但另一个 Bug 记录却把这个属性称作 report_date。虽然两者都很清晰地表达了同样的意思。

既然如此，你又要如何计算每天的 Bug 数呢？

EAV/anti/count.sql

```
SELECT date_reported, COUNT(*) AS bugs_per_date
FROM (SELECT DISTINCT issue_id, attr_value AS date_reported
      FROM IssueAttributes
      WHERE attr_name IN ('date_reported', 'report_date'))
GROUP BY date_reported;
```

你又如何得知某条 Bug 记录没有用其他的名字来定义属性呢？你又如何得知某条 Bug 记录没有将同一个属性存储了两遍？你又要如何阻止这样的错误发生？

有一个解决方案是将 attr_name 列声明成一个外键，并指向一张存储着所有可能出现的属性名的表。然而，这不支持在运行时定义的各种属性，即使那是 EAV 设计的一个非常普遍的使用方式。

6.2.7 重组列

当数据是存储在一张传统的表中时，从 Issues 表中获取一整行记录，并得到一个议题的所有属性是个很平常的需求。

issue_id	date_reported	stats	优先级	描　述
1234	2009-06-01	NEW	HIGH	存储无法运行

由于每个属性在 IssueAttributes 表里都存储在独立的行中，要想像上面那样按行获取所有这些属性就需要执行一个联结查询，并将结果合并成行。同时，必须在写查询语句的时候就知道所有的属性名称。下面的查询语句重组了上表展示的行：

EAV/anti/reconstruct.sql

```
SELECT i.issue_id,
  i1.attr_value AS "date_reported",
  i2.attr_value AS "status",
  i3.attr_value AS "priority",
  i4.attr_value AS "description"
```

```
FROM Issues AS i
  LEFT OUTER JOIN IssueAttributes AS i1
    ON i.issue_id = i1.issue_id AND i1.attr_name = 'date_reported'
  LEFT OUTER JOIN IssueAttributes AS i2
    ON i.issue_id = i2.issue_id AND i2.attr_name = 'status'
  LEFT OUTER JOIN IssueAttributes AS i3
    ON i.issue_id = i3.issue_id AND i3.attr_name = 'priority';
  LEFT OUTER JOIN IssueAttributes AS i4
    ON i.issue_id = i4.issue_id AND i4.attr_name = 'description';
WHERE i.issue_id = 1234;
```

你必须使用外联结来进行查询，因为如果在所查询的这些属性中有任何一个不在 IssueAttributes 表中出现，则内联结会导致整个查询返回空记录。随着属性的数量不断增多，联结的数量也不断增长，查询的开销也成指数级地增长。

6.3　如何识别反模式

如果你听到项目团队发出了如下的疑问，很有可能就是使用了 EAV 反模式。

- "数据库不需要修改元数据就可以扩展。你还可以在运行时定义新的属性。"
  关系数据库不支持这种程度的灵活性。当某人声称能够设计一个可以任意扩展的数据库时，他基本上用的就是 EAV 的设计。
- "查询时我能用的最大数量的联结是多少？"
  如果你需要一个支持如此多联结的查询，并且联结的数量可能会达到数据库的限制时，你的数据库的设计可能是有问题的。而 EAV 的设计很有可能会导致这样的问题。
- "我想象不出怎么为我们的电子商务平台生成报告。我们需要雇一个顾问来做这事。"
  似乎很多现有的数据库驱动的软件使用 EAV 的设计来支持其强大的自定义能力，而这使得很多普通的报表查询变得极度复杂甚至不切实际。

6.4　合理使用反模式

在关系数据库中很难为 EAV 这个反模式正名，因为这就不得不放弃关系型范式的太多优点。但这不影响在某些程序中合理地使用这种设计来支持动态属性。

大多数应用程序仅仅在有限的几张表甚至于仅一张中需要存储无范式的数据，而其他的数据需求适用于标准的表设计。如果你明白在你的项目中使用 EAV 设计的风险和你要做的额外工作，并且谨慎地使用它，它的副作用会变得较小。但请一定要记住，那些富有经验的数据库顾问给出的报告显示，使用 EAV 设计的系统在一年以内就会变得极其笨重。

如果你有非关系数据管理的需求，最好的答案是使用非关系技术。这是一本关于 SQL 的书，而不是关于 SQL 选择的问题，因此我会简单地列出一些相关的技术。

- Berkeley DB 是一个流行的 Key-Value 存储服务，非常容易被整合进入大部分程序。
  http://www.oracle.com/technology/products/berkeley-db/
- Cassandra 是一个分布式的面向列的数据库，由 Facebook 开发，并提交给了 Apache 项目。
  http://incubator.apache.org/cassandra/
- CouchDB 是一个面向文档的数据库——一个分布式的 Key-Value 存储系统，使用 JSON 编码数据。
  http://couchdb.apache.org/
- Hadoop 和 HBase 组装了一个开源的 DBMS，借助于 Google 的 MapReduce 算法为分布式大数据量查询提供分结构数据存储。
  http://hadoop.apache.org/
- MongoDB 是一个像 CouchDB 一样的面向文档的数据库。
  http://www.mongodb.org/
- Redis 是一个文件导向的内存数据库。
  http://code.google.com/p/redis/
- Tokyo Cabinet 是一个 Key-Value 存储结构，结合了 POSIX DBM、GNU GDBM 或 Berkeley DB。
  http://1978th.net/

很多其他的非关系数据库项目也在不断地涌现。然而，在传统数据库中使用 EAV 设计的劣势也体现在这些非关系数据库上。当元数据不具有固定格式时，再简单的查询都会变得非常困难。上层应用就需要花费更多的时间、精力来组织数据结构。

6.5 解决方案：模型化子类型

如果 EAV 对于你的程序而言是正确的选择，你仍然需要在执行这个设计之前重新审视一遍。通过对列进行一些虽然老式但很好的分析，很可能会发现你的项目的数据可以更方便地被模型化到一个传统的表里，同时也提供了更保险的数据完整性支持。

除去使用 EAV，还有好几个方法来存储这样的数据。当子类型数量有限时，大多数解决方案都能很好地工作，并且你知道每个子类型的属性。哪个解决方案最合适依赖于你查询数据的方式，因此你应该具体案例具体分析。

6.5.1 单表继承

最简单的设计是将所有相关的类型都存在一张表中，为所有类型的所有属性都保留一列。同时，使用一个属性来定义每一行表示的子类型。在这个例子中，这个属性称作 `issue_type`。对于所有的子类型来说，既有一些公共属性，但同时又有一些子类型特有属性。这些子类型特有属

性列必须支持空值，因为根据子类型的不同，有些属性并不需要填写，从而对于一条记录来说，那些非空的项会变得比较零散。

这个设计的名字来源于 Martin Flower 的一本著作：*Patterns of Enterprise Application Architecture*[Fow03]。

EAV/soln/create-sti-table.sql

```
CREATE TABLE Issues (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  issue_type       VARCHAR(10),  -- BUG or FEATURE
  severity         VARCHAR(20),  -- only for bugs
  version_affected VARCHAR(20),  -- only for bugs
  sponsor          VARCHAR(50),  -- only for feature requests
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

当程序需要加入新对象时，必须修改数据库来适应这些新对象。又由于这些新对象具有一些和老对象不同的属性，因而必须在原有表里增加新的属性列。可能会遇到一个很实际问题，就是每张表的列的数量是有限制的。

单表继承的另一个限制就是没有任何的元信息来记录哪个属性属于哪个子类型。在你的程序中，假如你知道有些属性并不适用于一个特定的行所表示的子类型对象，那么就可以忽略它们。但必须手动地跟踪哪些属性适用于哪些子类型。即使我们知道如果能用元数据在数据库中定义这些会更好，但也无能为力。

当数据的子类型很少，以及子类型特殊属性很少，并且你需要使用 Active Record 模式来访问单表数据库时，单表继承模式是最佳选择。

6.5.2 实体表继承

另一个解决方案是为每个子类型创建一张独立的表。每个表包含那些属于基类的共有属性，同时也包含子类型特殊化的属性。这个设计的名字来源于 Martin Fowler 的书。

EAV/soln/create-concrete-tables.sql

```
CREATE TABLE Bugs (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
```

```
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  severity         VARCHAR(20), -- only for bugs
  version_affected VARCHAR(20), -- only for bugs
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE FeatureRequests (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  sponsor          VARCHAR(50),  -- only for feature requests
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

实体继承设计相比于单表继承设计的优势在于提供了一种方法，让你能阻止在一行内存储一些和当前子类型无关的属性。如果你引用一个并不存在于这张表中的属性列，数据库会自动提示你错误。比如，severity 列并不在 FeatureRequests 表中：

EAV/soln/insert-concrete.sql

```
INSERT INTO FeatureRequests (issue_id, severity) VALUES ( ... ); -- ERROR!
```

另一个使用实体继承表设计的好处便是，不用像在单表继承设计里那样使用额外的属性来标记子类型。

然而，很难将通用属性和子类特有的属性区分开来。因此，如果将一个新的属性增加到通用属性中，必须为每个子类表都加一遍。

没有元数据标记这些存储在自己表中的子类型相互之间有什么关系。那意味着，如果一个新来的程序员查看这些表定义，他只会注意到所有子类型的表中都有一些重复的列，但元信息没有告诉他任何有关这些表之间的关系，或者是否仅仅由于某种巧合，才使得这些表长得如此相似。

如果你希望不考虑子类型而在所有对象中进行过滤查找，问题会变得很复杂。如果想要将这件事情变得简单一点，就需要创建一个视图联合这些表，仅选择公共的列。

EAV/soln/view-concrete.sql

```
CREATE VIEW Issues AS
  SELECT b.*, 'bug' AS issue_type
  FROM Bugs AS b
   UNION ALL
```

```
  SELECT f.*, 'feature' AS issue_type
  FROM FeatureRequests AS f;
```

当你很少需要一次性查询所有子类型时，实体继承表设计是最好的选择。

6.5.3 类表继承

第三个解决方案模拟了继承，把表当成面向对象里的类。创建一张基类表，包含所有子类型的公共属性。对于每个子类型，创建一个独立的表，通过外键和基类表相连。这个设计的名称同样来自于 Martin Fowler 的书。

EAV/soln/create-class-tables.sql

```
CREATE TABLE Issues (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE Bugs (
  issue_id         BIGINT UNSIGNED PRIMARY KEY,
  severity         VARCHAR(20),
  version_affected VARCHAR(20),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);

CREATE TABLE FeatureRequests (
  issue_id         BIGINT UNSIGNED PRIMARY KEY,
  sponsor          VARCHAR(50),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);
```

基类表和子类表之间一对一的关系由元数据来确保，因为子类型表的外键也同样是主键，因而就必须是唯一的。这个解决方案提供了一个高效的方法来查询所有的记录，因为你仅仅查询基类的属性。一旦你找到了合适的记录，就可以通过查询对应的子类型表来获取子类型特殊化的属性。

你不需要了解在基类表中的行表示的是哪个子类型，如果仅有很少的子类型，那么可以写一个联结查询来一次性获取所有的记录，产生一个像单表继承设计里的那样稀疏结果集。当这个子类型不具有某个属性时，其值是空的。

EAV/soln/select-class.sql

```
SELECT i.*, b.*, f.*
```

```
FROM Issues AS i
  LEFT OUTER JOIN Bugs AS b USING (issue_id)
  LEFT OUTER JOIN FeatureRequests AS f USING (issue_id);
```

这也是定义一个视图的好方法。

当你经常要查询所有子类型时这个设计是最佳选择，引用这些公共列就行了。

6.5.4 半结构化数据模型

如果你有很多子类型或者你必须经常地增加新的属性支持，那么可以使用一个 BLOB 列来存储数据，用 XML 或者 JSON 格式——同时包含了属性的名字和值。Martin Fowler 称这个模式为：序列化大对象块（Serialized LOB）。

EAV/soln/create-blob-tables.sql

```
CREATE TABLE Issues (
  issue_id          SERIAL PRIMARY KEY,
  reported_by       BIGINT UNSIGNED NOT NULL,
  product_id        BIGINT UNSIGNED,
  priority          VARCHAR(20),
  version_resolved  VARCHAR(20),
  status            VARCHAR(20),
  issue_type        VARCHAR(10),   -- BUG or FEATURE
  attributes        TEXT NOT NULL, -- all dynamic attributes for the row
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

这个设计的优势之处就在于其优异的扩展性。你可以在任何时候，将新的属性添加到 blob 字段中。每行存储一个完整的属性集合，因此你可以有尽可能多的子类型，有多少行就可以有多少个。

相应地，该设计的缺点就是在这样的一个结构中，SQL 基本上没有办法获取某个指定的属性。你不能在一行 blob 字段中简单地选择一个独立的属性，并对其进行限制、聚合运算、排序等其他操作。你必须获取整个 blob 字段结构并通过程序去解码并且解释这些属性。

当你不能将需求和设计限制在一个有限的子类型集合中，或者当你需要绝对的灵活性以在任何时间调整属性时，这个方案就是最佳选择。

6.5.5 后处理

遗憾的是，有时你不得不使用 EAV 设计，比如你接手了一个项目，但不能改变它的原始设计，或者你的公司获得了一个第三方的软件，并且恰巧使用的是 EAV。如果是这样的情况，请牢记在 6.2 节中我们所描述的那些问题，从而你可以有所准备并且计划好额外的工作来让这个设计工作良好。

综上所述，别尝试像在传统表中那样写查询语句，将实体当成单行数据读取。取而代之的是，查询和实体关联的属性并且将这些行组合在一起，就像它们存储的结构一样。

EAV/soln/post-process.sql

```
SELECT issue_id, attr_name, attr_value
FROM IssueAttributes
WHERE issue_id = 1234;
```

查询的结果应该如下表：

issue_id	attr_name	attr_value
1234	date_reported	2009-06-01
1234	description	Saving does not work
1234	priority	HIGH
1234	product	Open RoundFile
1234	reported_by	Bill
1234	severity	loss of functionality
1234	status	NEW

这个查询对你来说写起来很容易，对数据库来说执行起来也很容易。即使当你写这个查询的时候并不知道有多少相关属性，它依旧会返回和这个事件相关的所有属性。

要使用这种格式的结果，你需要在程序中写一段代码来遍历结果集中的每一行记录，并且设置程序中对象的属性。下面有个 PHP 的代码范例：

EAV/soln/post-process.php

```php
<?php

$objects = array();

$stmt = $pdo->query(
    "SELECT issue_id, attr_name, attr_value
     FROM IssueAttributes
     WHERE issue_id = 1234");

while ($row = $stmt->fetch()) {
  $id    = $row['issue_id'];
  $field = $row['attr_name'];
  $value = $row['attr_value'];
  if (!array_key_exists($id, $objects)) {
    $objects[$id] = new stdClass();
  }

  $objects[$id]->$field = $value;
}
```

这看上去有太多的工作要做，但这就是使用像 EAV 这样的一个系统套系统结构所造成的必然结果。

SQL 已经提供了一个方法来明确地定义属性——在明确的列中。使用 EAV 设计，你让 SQL 使用新的方法来定义属性，因此 SQL 对于这种方法的支持是如此笨拙和低效也不足为奇了。

为元数据使用元数据。

第 7 章 多态关联

的确，有些人是两面派。

► 稻草人，《绿野仙踪》

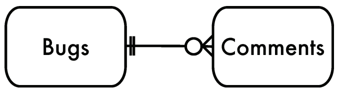


图 7-1　简单关系

让我们允许用户对 Bug 记录进行评论。一个给定的 Bug 可能会有很多评论，但任何的评论都只针对一个 Bug 记录。因此，在 Bug 和评论之间是一对多的关系。这种简单的关系如图 7-1 所示，接下来的 SQL 脚本显示如何创建这张表。

Polymorphic/intro/comments.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author_id    BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (author_id) REFERENCES Accounts(account_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

但是，可以进行评论的表可能会有两张，Bugs 和 FeatureRequests 是类似的实体，尽管你可能分开存储它们（参见 6.5 节）。你想要在单表中存储所有的评论，不关心它们的类型——无论是 Bug 还是新特性——但你不能声明一个指向多张表的外键。如下的定义是无效的：

Polymorphic/intro/nonsense.sql

```
  ...
  FOREIGN KEY (issue_id)
      REFERENCES Bugs(issue_id) OR FeatureRequests(issue_id)
);
```

有些开发人员还尝试着写如下的 SQL 语句查询多张表，却是无效的：

Polymorphic/intro/nonsense.sql

```
SELECT c.*, i.summary, i.status
FROM Comments AS c
JOIN c.issue_type AS i USING (issue_id);
```

SQL 不支持按行联结不同的表。SQL 的语法要求提交查询时就明确写明所有的表名。查询过程中表名不能修改。这样的情况问题出在哪里，要怎么解决呢？

7.1 目标：引用多个父表

在“绿野仙踪”里，当多萝西询问稻草人她应该走哪条路才能到翡翠城时，稻草人指给了她不确定的方向。本应该是非常简单的问题，但当稻草人一次指了两条路给她时，多萝西迷惑起来。

图 7-2 描绘了在实体关系中的这种令人迷惑的联系。子表中的外键“分叉”了，因此 Comments 表中的一条记录即可能匹配 Bugs 表中的某条记录，也可能匹配于 FeatureReqeuests 表中的某条记录。

图中的弧线表明这是一个排他选择：一个给定的评论只能引用一个 Bug 或者一个特性。

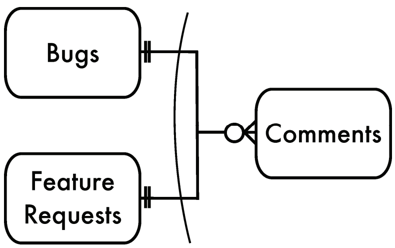


图 7-2 多态关联

7.2 反模式：使用双用途外键

有一个解决方案已经流行到足以正式命名了，那就是：多态关联。有时候也叫做杂乱关联，因为它可以同时引用多个表。

7.2.1 定义多态关联

为了使多态关联能够正常工作，在 issue_id 这个外键之外你必须再添加一列，这个额外的列记录了当前行所引用的表名。在这个例子中，这个额外的列称为 issue_type，取值范围是 Bugs 或者 FeatureRequests。

Polymorphic/anti/comments.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  issue_type   VARCHAR(20),     -- "Bugs" or "FeatureRequests"
  issue_id     BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME,
  comment      TEXT,
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

你立刻就可以发现一个区别：issue_id 上的外键定义不见了。事实上，由于一个外键必须指定一个确切的表，使用多态关联就意味着无法在元数据中定义这样的关系。因而，就没有任何保障数据完整性的手段来确保 Comments.issue_id 中的值在其父表中存在。

同样地，没有元数据来确保 Comments.issue_type 中的值确实对应于数据库中存在的一张表。

混合数据与元数据

你可能已经注意到了多态关联和前一章的 EAV 反模式有着相似的特征。在这两个反模式中，元数据对象的名字是存储在字符串中的。在 EAV 中，属性列的名字是以字符串的格式存储在 attr_name 列中。在多态关联中，父表的名字是存储在 issue_type 列中的。有时这样的设计被称为：混合数据与元数据。第 8 章会有更详细的概念解释。

7.2.2 使用多态关联进行查询

在 Comments 表中的一个 issue_id 可能同时出现在 Bugs 和 FeatureRequests 这两张父表中，或者只出现在其中一张表里。因此，在使用 issue_type 进行联结查询时，就必须格外谨慎，必须确保不会出现明明是要查找 Bugs 的评论，却获得了 FeatureRequests 的评论这种情况。

比如，如下的查询目的在于获取 id 为 1234 的这个 Bug 的所有评论：

Polymorphic/anti/select.sql

```
SELECT *
FROM Bugs AS b JOIN Comments AS c
  ON (b.issue_id = c.issue_id AND c.issue_type = 'Bugs')
WHERE b.issue_id = 1234;
```

当所有的 Bug 记录都是存在单个的 Bugs 表中时，上面的这个查询能够正常地工作，但当 Comments 同时和 Bugs 表以及 FeatureRequests 表相关联时，就会出现问题。SQL 的语法规定，在联结查询时必须指明所有需要查询的表，没办法在查询过程中根据 Comments.issue_type 的值来切换不同的表。

要查找一条给定的评论对应的 Bug 记录或者特性需求，需要执行一条同时外联 Bugs 和 FeatureRequests 两张表的查询。仅有一张表满足这个查询需求，因为联结查询的条件依赖于 Comment.issue_type 中的值。使用外联结意味着结果中那些来自于非匹配的字段将为空值。

Polymorphic/anti/select.sql

```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN Bugs AS b
    ON (b.issue_id = c.issue_id AND c.issue_type = 'Bugs')
  LEFT OUTER JOIN FeatureRequests AS f
    ON (f.issue_id = c.issue_id AND c.issue_type = 'FeatureRequests');
```

结果看起来类似下面这样。

c.comment_id	c.issue_type	c.issue_id	c.comment	b.issue_id	f.issue_id
6789	Bugs	1234	It crashes!	1234	NULL
9876	Feature…	2345	Great idea!	NULL	2345

7.2.3 非面向对象范例

在 Bugs 和 FeatureRequests 的例子中，这两张表代表了模型相关的子类型。多态关联也同样可以用在那些父表间完全无关的设计中。比如，在一个电子商务数据库中，Users（用户）和 Orders（订单）这两张表可能都和 Addresses（地址）相关，如图 7-3 所示。

Polymorphic/anti/addresses.sql

```
CREATE TABLE Addresses (
  address_id   SERIAL PRIMARY KEY,
  parent       VARCHAR(20),     -- "Users" or "Orders"
  parent_id    BIGINT UNSIGNED NOT NULL,
  address      TEXT
);
```

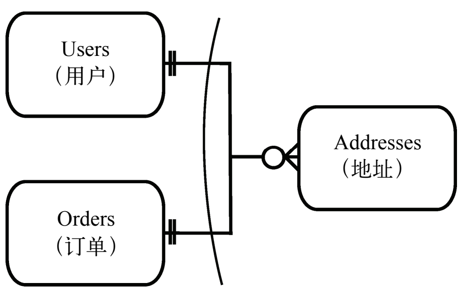


图 7-3 地址多态关联

在这个例子中，Addresses 表有一个多态列，对于给定的一条地址记录，其父表的值为 Users

或者 Orders，两者二选其一。一个给定的地址不能既和用户相关，又和订单相关，即使订单可能是由一个用户创建然后邮寄给他自己的。

此外，如果一个用户的收货地址也是账单地址，你应该在 Addresses 表中将其区分开来;同样地，任何其他的父表都需要留意 Addresses 表中不同的地址字段的特殊用处。这些说明信息将像野草一样疯长。

Polymorphic/anti/addresses.sql

```
CREATE TABLE Addresses (
  address_id   SERIAL PRIMARY KEY,
  parent       VARCHAR(20),     -- "Users" or "Orders"
  parent_id    BIGINT UNSIGNED NOT NULL,
  users_usage  VARCHAR(20),     -- "billing" or "shipping"
  orders_usage VARCHAR(20),     -- "billing" or "shipping"
  address      TEXT
);
```

7.3 如何识别反模式

如果你听到类似下面的这些说法，那有可能就使用了多态关联的反模式了。

- “这种标记架构可以让你将标记（或者其他属性）和数据库中的任何其他资源联系起来。”
  就像 EAV 的设计一样，你应该怀疑任何声称有无限扩展性的设计。
- “你不能在我们的数据库设计中声明外键。”
  这是另一个危险信号。外键是关系数据库的基本特征，一个没有适当的引用完整性的设计会有很多的问题。
- “entity_type 这列是干嘛的？哦，那个列是用来告诉你这条记录的其他列是和什么东西相关的。”
  任何外键都强制一张表中所有的行引用同一张表。

Ruby on Rails 的框架通过声明带有:polymorphic 属性的 Active Record 类来支持多态关联。比如，你可以使用如下方式将 Comments 和 Bugs 以及 FeatureRequests 关联起来。

Polymorphic/recog/commentable.rb

```
class Comment < ActiveRecord::Base
  belongs_to :commentable, :polymorphic => true
end

class Bug < ActiveRecord::Base
  has_many :comments, :as => :commentable
end
```

```
class FeatureRequest < ActiveRecord::Base
  has_many :comments, :as => :commentable
end
```

Java Hibernate 框架使用大量的模式定义来支持多态关联。

7.4 合理使用反模式

你应该尽可能地避免使用多态关联——应该使用外键约束等来确保引用完整性。多态关联通常过度依赖上层程序代码而不是数据库的元数据。

当你使用一个面向对象的框架（诸如 Hibernate）时，多态关联似乎是不可避免的。这种类型的框架通过良好的逻辑封装来减少使用多态关联的风险。如果你选择了一个成熟、有信誉的框架，那可以相信框架的作者已经完整地实现了相关的逻辑代码，不会造成错误。但如果你不使用框架而自己从头开始实现，那就真有重新发明轮子之嫌了。

7.5 解决方案：让关系变得简单

既要避免多态关联的缺点，又同时支持你所需要的数据模型，最好的选择是重新设计数据库。接下来几节介绍的解决方案能够完全满足我们所需的数据关系，同时又使用数据库的元数据来确保数据及引用的完整性。

7.5.1 反向引用

当你看清楚问题的根源时，解决方案将变得异常的简单：多态关联是一个反向关联。

7.5.2 创建交叉表

Comments 表中的外键无法同时引用多张父表，因而，我们使用多个外键同时引用 Comments 表即可。为每个父表创建一张独立的交叉表，每张交叉表同时包含一个指向 Comments 的外键和一个指向对应父表的外键，如图 7-4 所示。

Polymorphic/soln/reverse-reference.sql

```
CREATE TABLE BugsComments (
  issue_id    BIGINT UNSIGNED NOT NULL,
  comment_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES Bugs(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);

CREATE TABLE FeaturesComments (
  issue_id    BIGINT UNSIGNED NOT NULL,
```

```
  comment_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES FeatureRequests(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);
```

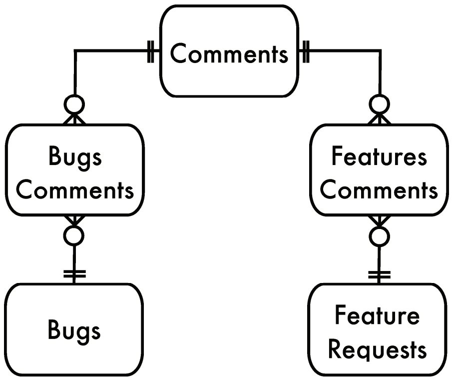


图 7-4 反向多态关联

这个解决方案移除了对 Comments.issue_type 列的依赖。现在，元数据可以确保数据完整性了，从而不再依赖于应用程序代码来维护数据间的关系。

7.5.3 设立交通灯

这个方案的潜在问题是可以加入一些你可能不希望出现的关系。交叉表通常是多对多关系的模型，因而这个设计允许一个给定的评论同时和多个 Bug 或者多个特性记录相关。然而，你可能是希望每条评论都只涉及一个 Bug 或者一个特性需求。我们可以通过在每张交叉表的 comment_id 列上声明一个 UNIQUE 的约束来尽可能地支持这样的规则。

Polymorphic/soln/reverse-unique.sql

```
CREATE TABLE BugsComments (
  issue_id    BIGINT UNSIGNED NOT NULL,
  comment_id  BIGINT UNSIGNED NOT NULL,
  UNIQUE KEY (comment_id),
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES Bugs(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);
```

这样就能确保一个给定的评论在一张交叉表中只能出现一次，从而确保其只能和一个 Bug 或者一个特性相关。然而，元数据并不能保证一个给定的评论不在多张交叉表中被关联。如果这不是你所希望的，则只能交由上层的应用程序代码来完成了。

7.5.4　双向查找

我们可以方便地使用交叉表查询一个给定的 Bug 或者特性需求的评论。

Polymorphic/soln/reverse-join.sql

```
SELECT *
FROM BugsComments AS b
  JOIN Comments AS c USING (comment_id)
WHERE b.issue_id = 1234;
```

我们也可以使用一个外联结查询两张交叉表来获得一条评论所对应的 Bug 或者特性需求记录。虽然在查询时仍不得不给出所有相关的表名，但已经比使用多态关联时简单不少。同样地，使用交叉表还能获得使用多态关联所不能提供的引用完整性支持。

Polymorphic/soln/reverse-join.sql

```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN (BugsComments JOIN Bugs AS b USING (issue_id))
    USING (comment_id)
  LEFT OUTER JOIN (FeaturesComments JOIN FeatureRequests AS f USING (issue_id))
    USING (comment_id)
WHERE c.comment_id = 9876;
```

7.5.5　合并跑道

某些情况下你并没有使用之前描述的多表结构，而将多张父表都存在同一张表中（详见 6.5 节），你可以使用如下的两种方法来获取你所需要的结果。

首先我们来看下面这个使用 UNION 的查询：

Polymorphic/soln/reverse-union.sql

```
SELECT b.issue_id, b.description, b.reporter, b.priority, b.status,
    b.severity, b.version_affected,
    NULL AS sponsor
  FROM Comments AS c
    JOIN (BugsComments JOIN Bugs AS b USING (issue_id))
      USING (comment_id)
  WHERE c.comment_id = 9876;
UNION
  SELECT f.issue_id, f.description, f.reporter, f.priority, f.status,
    NULL AS severity, NULL AS version_affected,
    f.sponsor
  FROM Comments AS c
    JOIN (FeaturesComments JOIN FeatureRequests AS f USING (issue_id))
      USING (comment_id)
  WHERE c.comment_id = 9876;
```

如果你的程序已经确定将每条评论只和一张父表关联，那么这个查询是能够保证每次只返回一条记录的。由于 UNION 查询只有在列的数量和数据类型都一样时，才能将查询结果合并，因此你必须要为每一个父表不同的列提供一个 null 占位符。在 UNION 的查询中，你还必须用同样的顺序排列两次查询的列。

另一种方法，可以先看一下下面这个使用 SQL 的 COALESCE()函数的查询。这个函数返回第一个非空的结果。由于我们使用了外联结查询，一条和需求相关并且在 Bugs 表中没有匹配项的评论，所有 b.*的列都将被赋值为空。同样地，一条和 Bug 相关的评论，所有 f.*的列都将被默认地填入空值。用简洁的方式列出针对某一父表专有的列：凡是该记录和某一父表无关，则那些字段均返回 null。

Polymorphic/soln/reverse-coalesce.sql

```
SELECT c.*,
  COALESCE(b.issue_id,    f.issue_id   ) AS issue_id,
  COALESCE(b.description, f.description) AS description,
  COALESCE(b.reporter,    f.reporter   ) AS reporter,
  COALESCE(b.priority,    f.priority   ) AS priority,
  COALESCE(b.status,      f.status     ) AS status,
  b.severity,
  b.version_affected,
  f.sponsor
FROM Comments AS c
  LEFT OUTER JOIN (BugsComments JOIN Bugs AS b USING (issue_id))
    USING (comment_id)
  LEFT OUTER JOIN (FeaturesComments JOIN FeatureRequests AS f USING (issue_id))
    USING (comment_id)
WHERE c.comment_id = 9876;
```

这几个查询的方案都比较复杂，因此，比较好的方式是通过它们创建数据库视图，然后就可以在你的应用程序中较为简便地使用。

7.5.6 创建共用的超级表

在面向对象的多态机制中，两个继承自同一个父类的子类型可以使用相似的方式来使用。在 SQL 中，多态关联这个反模式遗漏了一个关键实质：共用的父对象。我们可以通过创建一个基类表，并让所有的父表都从这个基类表扩展出来的方法来解决这个问题（参考 6.5 节）。在 Comments 子表中添加一个指向基类表的外键，并且不再需要 issue_type 列。这个解决方案的实体图可以用图 7-5 表示。

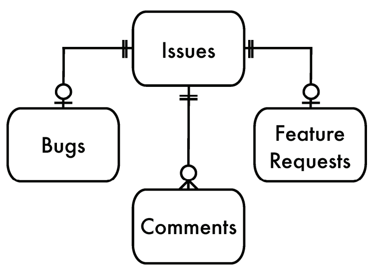


图 7-5　评论和基类表的联合

Polymorphic/soln/super-table.sql

```
CREATE TABLE Issues (
```

```
  issue_id     SERIAL PRIMARY KEY
);

CREATE TABLE Bugs (
  issue_id     BIGINT UNSIGNED PRIMARY KEY,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  . . .
);

CREATE TABLE FeatureRequests (
  issue_id     BIGINT UNSIGNED PRIMARY KEY,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  . . .
);
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  issue_id     BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME,
  comment      TEXT,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id),
);
```

请注意 Bugs 和 FeatureRequests 表的主键也同时是外键。它们引用了 Issues 表所维护的代理主键，而不用自己创建它们。

给定一个评论，你可以通过一个相对简单的查询就获取对应的 Bug 记录或者需求记录。而不再需要在查询中包含 Issues 表，除非你将一些属性定义在那张表里。同样地，由于 Bugs 表的主键和它的祖先 Issues 表中的值是一样的，你可以直接对 Bugs 表和 Comments 表进行联结查询。也可以对两张没有外键约束直接关联的表进行联结查询，只要对应的列的信息是可比较的即可。

Polymorphic/soln/super-join.sql

```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN Bugs AS b USING (issue_id)
  LEFT OUTER JOIN FeatureRequests AS f USING (issue_id)
WHERE c.comment_id = 9876;
```

对于一个指定 Bug，你同样可以轻易地读出它的评论。

Polymorphic/soln/super-join.sql

```
SELECT *
FROM Bugs AS b
  JOIN Comments AS c USING (issue_id)
WHERE b.issue_id = 1234;
```

更重要的是如果你使用了像 `Issues` 这样的祖先表，就可以依赖外键来确保数据的完整性。

在每个表与表的关系中，都有一个引用表和一个被引用表。

第8章 多列属性

超群和荒谬的分界线往往非常模糊，很难明确地区分。

▶ 托马斯·潘恩

我已经数不清创建过多少次存储联系人信息的表了。每次都有一些共同的字段，比如名字、称呼、地址、公司等。

电话号码有一点棘手。通常人们会同时拥有多个号码：家庭电话，工作电话，传真号码以及手机号码。在联系人信息表中，分4列来存储这些信息是很简单的方法。

但对于其他号码呢？这个人的助理的号码、另一个移动电话的号码，或者外地办事处的全然不同的电话号码，甚至还有些无法预计的分类。我可以为这些并不常有的情况创建更多的列，但这看上去很笨拙，因为加了很多很少使用的字段。而且，到底要加多少这样的列才够呢？

8.1 目标：存储多值属性

这是和第2章一样的目标：一个属性看上去虽然只属于一张表，但同时可能会有多个值。之前我们已经看到，将多个值合并在一起并用逗号分隔导致难以对数据进行验证，难以读取或者改变单个值，同时也对聚合公式（诸如统计不同值的数量）非常不友好。

我们将使用一个新的例子来说明这一反模式。我们将让这个Bug数据库允许加入标签，因而就可以以此来分类Bug。某些Bug可能是由它们所影响的软件子系统，比如打印、报表或者邮件来分类的；另一些Bug可能是由它们的类型来分类的，比如一个造成程序崩溃的Bug会被标记为crash，也可以标记为performance来说明性能问题，当然还可以标记cosmetic来说明用户界面的颜色选择不好。

这个给Bug打标签的特性必须支持多标签，因为标签并不会相互排斥。一个Bug可能同时影响到多个子系统，也有可能会影响到子系统的某个特性，比如“打印的性能”。

8.2 反模式：创建多个列

我们依旧需要考虑一个属性的多个值，但我们知道每列最好只存储一个值。在这张表中创建多个列，每个列只存储一个标签看上去很自然。

Multi-Column/anti/create-table.sql

```
CREATE TABLE Bugs (
  bug_id      SERIAL PRIMARY KEY,
  description VARCHAR(1000),
  tag1        VARCHAR(20),
  tag2        VARCHAR(20),
  tag3        VARCHAR(20)
);
```

当你将一个标签指定给一个 Bug 记录时，必须将这个标签存放于这三个列中的一个。其他未使用的列将保持空的状态。

Multi-Column/anti/update.sql

```
UPDATE Bugs SET tag2 = 'performance' WHERE bug_id = 3456;
```

bug_id	description	tag1	tag2	tag3
1234	Crashes while saving	crash	NULL	NULL
3456	Increase performance	printing	performance	NULL
5678	Support XML	NULL	NULL	NULL

在使用传统属性设计的时候，很简单的任务现在变得更复杂了。

8.2.1 查询数据

当根据一个给定标签查询所有 Bug 记录时，你必须搜索所有的三列，因为这个标签字符串可能存放于这三列中的任何一列。

比如，要获取被标记为 performance 的 Bug，需要使用如下的一个查询表达式：

Multi-Column/anti/search.sql

```
SELECT * FROM Bugs
WHERE tag1 = 'performance'
   OR tag2 = 'performance'
   OR tag3 = 'performance';
```

你还可能需要查找同时被标记为 performance 和 printing 的 Bug。要完成这样的查询，就需要写如下的查询语句。请注意必须正确地使用括号，因为 OR 操作比 AND 操作的优先级要低。

Multi-Column/anti/search-two-tags.sql

```
SELECT * FROM Bugs
WHERE (tag1 = 'performance' OR tag2 = 'performance' OR tag3 = 'performance')
  AND (tag1 = 'printing' OR tag2 = 'printing' OR tag3 = 'printing');
```

在多个列中查找一个值的语法是冗长乏味的。你可以通过一种非传统的方式来使用 IN，从而使得这个查询变得更加精简：

Multi-Column/anti/search-two-tags.sql

```
SELECT * FROM Bugs
WHERE 'performance' IN (tag1, tag2, tag3)
  AND 'printing'    IN (tag1, tag2, tag3);
```

8.2.2 添加及删除值

在这个列的集合中添加以及删除一个值也是有问题的。单纯地使用 UPDATE 语句来更新一列的值是不安全的，因为你无法得知到底哪一列是有值的（如果有的话）。你可能不得不将整行数据读取到前端程序中来分析。

Multi-Column/anti/add-tag-two-step.sql

```
SELECT * FROM Bugs WHERE bug_id = 3456;
```

举个例子，假设我们知道 tag2 是空的。于是写出如下的 UPDATE 语句。

Multi-Column/anti/add-tag-two-step.sql

```
UPDATE Bugs SET tag2 = 'performance' WHERE bug_id = 3456;
```

这么做，你就要面临多步操作间的数据同步问题，即有可能在你完成一次查询并且更新记录这两步操作之间，有另一个客户端也同时在执行相同的操作。根据更新操作执行顺序的不同，你或者另一个客户端将得到一个“更新冲突”的错误，或者一方覆盖了另一方的数据。你可以使用更复杂的 SQL 语句来避免这样的问题。

如下的表达式使用了 NULLIF()[①] 函数来将每一个等于指定值的列置为空。当传入的两个参数相等时，NULLIF()函数返回空。

Multi-Column/anti/remove-tag.sql

```
UPDATE Bugs
SET tag1 = NULLIF(tag1, 'performance'),
    tag2 = NULLIF(tag2, 'performance'),
    tag3 = NULLIF(tag3, 'performance')
WHERE bug_id = 3456;
```

如下的表达式将 performance 标签加到第一个空列中。然而，如果 3 列都不为空，这条语句将不对这条记录做任何修改，新的标签将不会被记录。同时，写这样的查询语句是非常繁琐耗时的，你必须要重复 performance 这个字符串 6 次！

Multi-Column/anti/add-tag.sql

```
UPDATE Bugs
SET tag1 = CASE
      WHEN 'performance' IN (tag2, tag3) THEN tag1
```

① NULLIF()在 SQL 中是一个标准函数，它被除 Informix 和 Ingres 以外的所有厂商支持。

```
    ELSE COALESCE(tag1, 'performance') END,
  tag2 = CASE
    WHEN 'performance' IN (tag1, tag3) THEN tag2
    ELSE COALESCE(tag2, 'performance') END,
  tag3 = CASE
    WHEN 'performance' IN (tag1, tag2) THEN tag3
    ELSE COALESCE(tag3, 'performance') END
WHERE bug_id = 3456;
```

8.2.3 确保唯一性

你可能并不希望同一个值出现在多个列中，但当你使用多值属性这个反模式时，数据库并不能阻止这样的情况发生。换而言之，很难阻止如下语句的执行：

Multi-Column/anti/insert-duplicate.sql

```
INSERT INTO Bugs (description, tag1, tag2, tag3)
  VALUES ('printing is slow', 'printing', 'performance', 'performance');
```

8.2.4 处理不断增长的值集

这个设计的另一个弱点在于三列可能并不够用。要保证每一列只存储一个值，你必须定义和一个Bug能支持的标签最大数一样多的列。在定义这张表的时候，你能预计多少标签数量是最大值吗？

有一个策略是暂时先猜测一个中等规模的量并在日后必要时对表进行扩展。大多数数据库允许你对已经存在的表进行重构，因此你可以增加 `Bug.tag4` 这个列，或者在需要时增加更多的列。

Multi-Column/anti/alter-table.sql

```
ALTER TABLE Bugs ADD COLUMN tag4 VARCHAR(20);
```

然而，这样的改变在以下三点上的开销是巨大的。

- 重构一张已经存在数据的表可能会导致锁住整张表，并阻止那些并发客户端的访问。
- 有些数据库是通过定义一张符合需求的新表，然后将现有数据从旧表中复制到新表中，再丢弃旧表的方式来实现重构表结构的。如果需要重构的表有很多数据，那转换过程将非常耗时。
- 在多列属性中增加了一列之后，你必须检查每一条相关的 SQL 语句，修改这些 SQL 语句以支持这些新加入的列。

Multi-Column/anti/search-four-columns.sql

```
SELECT * FROM Bugs
WHERE tag1 = 'performance'
  OR tag2 = 'performance'
  OR tag3 = 'performance'
  OR tag4 = 'performance'; -- you must add this new term
```

这是一个细致且费时的开发任务。如果你漏掉了任何需要修改的查询语句，就可能造成难以定位的错误。

8.3 如何识别反模式

反模式的模式

乱穿马路和多值属性这两个反模式都有一个共同的主线：这两个反模式都是解决同一个目标的解决方案——存储一个具有多个值的属性。

在乱穿马路的例子中，我们解决了多对多关系的存储。在本章中，我们将看到如何解决一对多的关系。不过需要明白的是，这两个反模式和两种数据间的关系有时是可以互用的。

如果用户界面上或者项目的设计文档中有任何属性可能具有多个值，并且这个值的数量具有一个固定的最大值，这可能意味着你使用了多值属性这个反模式。

诚然，有些属性的确可能为了某种目的，其候选值的数量具有最大值的限制，但通常情况下是没有这种限制的。如果这个限制是任意加上去的或者看上去不太合理，那可能就是因为使用了这个反模式了。

如果你听到有人说了下面这些话，那也是一个线索，提醒你可能使用了本章的反模式：

- "我们应该支持的标签数量的最大值是多少？"
  你需要决定为标签这样的多值属性定义多少列。
- "我要怎么才能在 SQL 查询中同时搜索多列？"
  如果你正在多列中查找一个给定的值，这是一个线索提示你应该存储多个具有同样逻辑属性的列。

8.4 合理使用反模式

在某些情况下，一个属性可能有固定数量的候选值，并且对应的存储位置和顺序都是固定的。比如，一个给定的 Bug 可能和多个用户账号相关，但每个关系的作用都是唯一的：一个是报告这个 Bug 的用户，另一个是修复这个 Bug 的开发人员，另一个是验证 Bug 修复状态的质量控制工程师。即使这几列里存储的值是相似的，它们的作用以及实际的业务逻辑都是不同的。

在 Bugs 表中定义三个不同的列来存储这三个属性是合理的。本章所描述的那些缺点在这里无关紧要，因为基本上这几列都是分开使用的。虽然有时仍旧需要对所有这三列数据进行查询，比如在一份"每个 Bug 都涉及哪些人"的报告里就需要这么做。既然这样的设计能够使得大部分

的查询用例都变得很简单，仅在有限数量的查询用例上表现得复杂还是能够被接受的。

另一种组织数据的方式是创建一张从属表来存储 Bugs 表和 Accounts 表之间的关系，同时在这张额外的表中增加一列来记录一个关系中的账号所表示的角色。然而，这样的设计可能会导致第 6 章 EAV 所描述的一些问题。

8.5 解决方案：创建从属表

如同我们在第 2 章中所看到的那样，最好的解决方案是创建一张从属表，仅使用一列来存储多值属性。将多个值存在多行中而不是多列里。同时，在从属表中定义一个外键，将这个值和 Bugs 表中的主记录关联起来。

Multi-Column/soln/create-table.sql

```
CREATE TABLE Tags (
  bug_id       BIGINT UNSIGNED NOT NULL
  tag          VARCHAR(20),
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

INSERT INTO Tags (bug_id, tag)
  VALUES (1234, 'crash'), (3456, 'printing'), (3456, 'performance');
```

当所有和 Bug 相关的标签都存储于一列中时，查找和一个给定标签相关的所有 Bug 就变得很直接了。

Multi-Column/soln/search.sql

```
SELECT * FROM Bugs JOIN Tags USING (bug_id)
WHERE tag = 'performance';
```

即使更复杂一点的查询，诸如“和两个标签相关的 Bug”，其查询代码也非常简单易读。

Multi-Column/soln/search-two-tags.sql

```
SELECT * FROM Bugs
  JOIN Tags AS t1 USING (bug_id)
  JOIN Tags AS t2 USING (bug_id)
WHERE t1.tag = 'printing' AND t2.tag = 'performance';
```

你可以使用比多值属性反模式更简单的方法来添加或移除数据间的关系——只要简单地添加或者删除从属表中的记录就行了。不再需要逐列检查是否有空列可以添加记录了。

Multi-Column/soln/insert-delete.sql

```
INSERT INTO Tags (bug_id, tag) VALUES (1234, 'save');

DELETE FROM Tags WHERE bug_id = 1234 AND tag = 'crash';
```

主键的约束能够保证不会有重复记录出现。一个给定的标签只能和一个给定的 Bug 关联一次，如果尝试重复插入，SQL 会告诉你错误。

每个 Bug 不再限制只能打三个标签了，不像在 `Bugs` 表中只能加三个 `tagN` 的列，现在只要有需要就可以一直加新的标签。

将具有同样意义的值存在同一列中。

第 9 章

元数据分裂

我不想再在船上看到这些东西。即使要付出我们所有的人作为代价我也不管，我就是要它们下船。

► 詹姆斯·柯克

我的妻子当 Java 及 Oracle PL/SQL 程序员已经好几年了。她提供了一个案例来展示什么样的数据库设计能使得工作变得更简单。

她的公司的销售部门使用的数据库中有一张叫做 `Customers` 的表，记录了和客户相关的信息，比如客户的业务类型，以及已经从这个客户那里获得了多少收入。

Metadata-Tribbles/intro/create-table.sql

```
CREATE TABLE Customers (
  customer_id   NUMBER(9) PRIMARY KEY,
  contact_info  VARCHAR(255),
  business_type VARCHAR(20),
  revenue       NUMBER(9,2)
);
```

但是销售部门需要按年来划分这些收入以便于跟踪每个客户的动态。他们决定增加一系列的列，每一列都按照年份来命名。

Metadata-Tribbles/intro/alter-table.sql

```
ALTER TABLE Customers ADD (revenue2002 NUMBER(9,2));
ALTER TABLE Customers ADD (revenue2003 NUMBER(9,2));
ALTER TABLE Customers ADD (revenue2004 NUMBER(9,2));
```

接着他们输入不完整的数据，只包含那些他们有兴趣跟踪的客户。大多数的行中，他们将这些收入字段置空。开发人员怀疑他们会不会在这些几乎用不到的列中存些别的数据？

每一年，他们都往这张表中增加一个新列。有一个专门的数据库管理员负责维护 Oracle 的表空间。因此，每一年，他们都需要开一系列的会议，协调数据迁移和重新组织表空间，并且增加新列。最终，他们浪费了一堆时间和金钱。

9.1 目标：支持可扩展性

随着数据量的增长，数据库的查询性能也会随之下降。即使查询结果可能只有很少的几千行，遍历表中积累的数据也可能使得整个查询的性能变得极差。即使使用了索引，但随着数据的增长，索引的作用变得非常有限。

本章的目标就是要优化数据库的结构来提升查询的性能以及支持表的平滑扩展。

9.2 反模式：克隆表与克隆列

在“星际迷航”中，tribbles 是一种被当做宠物饲养的小巧的、毛茸茸的动物。tribbles 最开始非常地吸引人，但很快它们就暴露出本性，开始了无节制的繁殖，然后管理泛滥的 tribbles 成了船上的一个严重问题。

你该把这些 tribbles 放哪儿？谁该为这些 tribbles 负责？要将所有的 tribbles 都收集起来要花多长时间？最终，柯克船长发现他的飞船及船员们无法正常工作，他不得不下令将“赶走 tribbles”作为头等大事来处理。

根据经验，我们知道查询一张表时，其性能只和这张表中数据的条数相关，越少的记录，查询速度越快。于是我们推导出一个常见错误的结论：无论要做什么，我们必须让每张表存储的记录尽可能少。这就导致了本章的反模式的两种表现形式。

- 将一张很长的表拆分成多张较小的表，使用表中某一个特定的数据字段来给这些拆分出来的表命名。
- 将一个列拆分成多个子列，使用别的列中的不同值给拆分出来的列命名。

但是你不能不劳而获：为了要达成减少每张表记录数的目的，你不得不创建一些有很多列的表，或者创建很多很多表。但在这两个方案中，你会发现随着数据量的增长，会有越来越多的表或者列，因为新的数据迫使你创建新的 Schema 对象。

混淆元数据和数据

请注意，通过将年份追加在基本表名之后，我们其实是将数据和元数据标识合并在了一起。

这和我们早先在 EAV 和多态关联反模式中看到的混合数据和元数据的方式正好相反。在那些案例中，我们将元数据标识（列名和表名）当做字符串存储。

在多列属性和元数据分裂模式中，我们将数据的值存储在列名或者表名中。如果你使用任何这些反模式，你只会得到更多的问题。

9.2.1　不断产生的新表

要将数据拆分到不同的表中，需要一个规则来定义哪些数据属于哪些表。比如，可以根据 date_reported 中的年份进行拆分：

Metadata-Tribbles/anti/create-tables.sql

```
CREATE TABLE Bugs_2008 ( . . . );
CREATE TABLE Bugs_2009 ( . . . );
CREATE TABLE Bugs_2010 ( . . . );
```

由于要将数据添加进数据库，于是根据要添加的数据的值选择合适的表就成了你的责任：

Metadata-Tribbles/anti/insert.sql

```
INSERT INTO Bugs_2010 (..., date_reported, ...) VALUES (..., '2010-06-01', ...);
```

让我们快进到 2011 年 1 月 1 日。你的程序在添加新的 Bug 报告时不断的报错，因为你忘记添加一张叫做 Bugs_2011 的新表。

Metadata-Tribbles/anti/insert.sql

```
INSERT INTO Bugs_2011 (..., date_reported, ...) VALUES (..., '2011-02-20', ...);
```

这意味着新的数据可能会需要新的元数据对象。这在 SQL 的设计中并不是常见的数据与元数据的关系。

9.2.2　管理数据完整性

假设你的老板打算计算这一年中 Bug 的数量，但他得到的数字并不正确。检查了相关数据和程序之后，你发现有一部分 2010 年的 Bug 被误写入了 Bugs_2009 这张表。如下的查询语句应该每次都返回空结果，如果不是的话，那么就有麻烦了：

Metadata-Tribbles/anti/data-integrity.sql

```
SELECT * FROM Bugs_2009
WHERE date_reported NOT BETWEEN '2009-01-01' AND '2009-12-31';
```

没有任何办法自动地对数据和相关表名做限制，但可以在每张表中都声明一个 CHECK 的约束：

Metadata-Tribbles/anti/check-constraint.sql

```
CREATE TABLE Bugs_2009 (
  -- other columns
  date_reported DATE CHECK (EXTRACT(YEAR FROM date_reported) = 2009)
);

CREATE TABLE Bugs_2010 (
  -- other columns
  date_reported DATE CHECK (EXTRACT(YEAR FROM date_reported) = 2010)
);
```

注意当你创建 Bugs_2011 时要记得修改 CHECK 约束里的值。如果你犯错了，就可能创建出一张拒绝接受正确值的表。

9.2.3 同步数据

某一天，你的客户支持部分析师想要改变 Bug 的日期。在数据库中存储的日期是 2010-01-03，但顾客实际是在一周以前，即 2009-12-27，使用传真报告的错误。你可以使用一个简单的 UPDATE 语句来修改这个值：

Metadata-Tribbles/anti/anomaly.sql

```
UPDATE Bugs_2010
SET date_reported = '2009-12-27'
WHERE bug_id = 1234;
```

但这个修正使得 Bugs_2010 表中的某一条记录变成了无效记录。你需要先从这张表中移除这条记录然后插入到另一张表中，在少数情况下，一个简单的 UPDATE 语句就会造成这样的异常状况。

Metadata-Tribbles/anti/synchronize.sql

```
INSERT INTO Bugs_2009 (bug_id, date_reported, ...)
  SELECT bug_id, date_reported, ...
  FROM Bugs_2010
  WHERE bug_id = 1234;

DELETE FROM Bugs_2010 WHERE bug_id = 1234;
```

9.2.4 确保唯一性

需要确保所有被分割出来的表中的主键都是唯一的。如果你需要从一张表中移动一条记录到另一张表中，需要保证被移动记录的主键值不会和目标表中的主键记录冲突。

如果使用一个支持序列化对象的数据库，那么可以使用一个简单的序列来确保主键值在所有的分割表中都是唯一的。对于那些只支持单表 ID 唯一的数据库产品来说，实现这样的功能变得很不优雅。你不得不定义一张额外的表来存储产品主键的值：

Metadata-Tribbles/anti/id-generator.sql

```
CREATE TABLE BugsIdGenerator (bug_id SERIAL PRIMARY KEY);

INSERT INTO BugsIdGenerator (bug_id) VALUES (DEFAULT);
ROLLBACK;

INSERT INTO Bugs_2010 (bug_id, . . .)
  VALUES (LAST_INSERT_ID(), . . .);
```

9.2.5 跨表查询

不可避免地，你的老板肯定会需要查询多张表中的数据。比如，他可能会要求查询所有的 Bug 总数，不管是哪一年报告的。你可以使用 UNION 将所有分割表联合起来得到一个重构过的 Bug 集合并将其作为一个衍生表进行查询：

Metadata-Tribbles/anti/union.sql

```
SELECT b.status, COUNT(*) AS count_per_status FROM (
  SELECT * FROM Bugs_2008
    UNION
  SELECT * FROM Bugs_2009
    UNION
  SELECT * FROM Bugs_2010 ) AS b
GROUP BY b.status;
```

年复一年，你创建了越来越多的表，比如 Bugs_2011，你需要不断地更新程序代码来引入这些新创建的表。

9.2.6 同步元数据

你的老板要求你在表中增加一列用以记录解决每个 Bug 所用的时间。

Metadata-Tribbles/anti/alter-table.sql

```
ALTER TABLE Bugs_2010 ADD COLUMN hours NUMERIC(9,2);
```

如果将表进行了拆分，那么这个新列仅仅被应用到你选择的那张表上。其他所有的表都不包含这个新列。

如果使用前一段描述的那个 UNION 查询所有的分割表，就会碰到一个新问题：当多张表具有相同的列时才能使用 UNION。如果多张表的列不完全相同，你就必须指明所有表都同时拥有的列，而不可以再使用“*”通配符。

9.2.7 管理引用完整性

如果一个关联表，比如 Comments 引用了 Bugs，这个关联表就不能声明一个外键了。一个外键必须指定单个表，但在这个案例中父表被拆分成很多个表。

Metadata-Tribbles/anti/foreign-key.sql

```
CREATE TABLE Comments (
  comment_id        SERIAL PRIMARY KEY,
  bug_id            BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs_????(bug_id)
);
```

分割表即使作为一张关联表而不是父表，同样可能引起一些问题。比如，Bugs.reported_by

引用了 Accounts 表。如果你想要查询一个给定的人所报告的所有 Bug，不管哪一年的，你需要使用像下面的这个查询：

Metadata-Tribbles/anti/join-union.sql

```
SELECT * FROM Accounts a
JOIN (
    SELECT * FROM Bugs_2008
    UNION ALL
    SELECT * FROM Bugs_2009
    UNION ALL
    SELECT * FROM Bugs_2010
  ) t ON (a.account_id = t.reported_by)
```

9.2.8 标识元数据分裂列

列也可能根据元数据分裂。你可以创建一个含有很多列的表，这些列按照它们的类别扩展，就像我们在本章最开始看到的那个故事一样。

我们的 Bug 数据库中可能碰到的另一个事例是：有一张表记录了项目指标的概要信息，其中的每一列存储了一个小计。具体来说，在如下这张表中，增加一个 bugs_fixed_2011 列只是时间问题：

Metadata-Tribbles/anti/multi-column.sql

```
CREATE TABLE ProjectHistory (
  bugs_fixed_2008  INT,
  bugs_fixed_2009  INT,
  bugs_fixed_2010  INT
);
```

9.3 如何识别反模式

如下的描述可能就是元数据分裂反模式在你的数据库中繁衍生长的暗示。

- “那么我们需要每……创建一张表（或者列）。”
  当你这样描述数据库时，说明你正根据某一列的值的范围拆分表。
- “数据库所支持的最大数量的表（或者列）是多少？”
  如果你的设计合理，大多数的数据库可以处理的表和列的数量比你所需要的多得多。如果你认为你的数据库设计可能会超过最大值，很明显应该重新考虑设计了。
- “我们终于发现为什么今早程序添加新记录失败了：我们忘记为新的一年添加新表了。”
  这是元数据分裂的普遍结果。当新的数据需要新的数据库对象时，你需要预先定义这些对象，否则就要接受这些不可预计的错误。
- “我要怎样同时查询很多张表？每张表的列都是一样的。”

如果你需要查询很多结构一样的表，就应该将数据全都存在一个表中，使用一个额外的属性列来分组数据。

- “我要怎样将表名作为一个变量传递？我在查询时需要根据年份动态地生成这些表名。”
  如果你的数据都在一张表里，那你根本不需要做这些事情。

9.4 合理使用反模式

手动分割表的一个合理使用场景是*归档数据*——将历史数据从日常使用的数据中移除。通常在过期数据的查询变得非常稀少的情况下，才会进行如此的操作。

如果你没有同时查询当前数据和历史数据的需求，将老数据从当前活动的表转移到其他地方是很合适的操作。

将数据归档到和当前表结构相兼容的新表中，既能支持偶尔做数据分析时的查询，同时也能让日常数据查询变得非常高效。

WordPress.com 使用的分区数据库设计

在 2009 年的 MySQL 大会上，我和 Barry Abrahamson 一起吃饭，他是 WordPress.com 的数据库架构师，而 WordPress.com 是一个很流行的博客服务。

Barry 说他开始做博客主机服务时，将所有客户的数据存在一个数据库中，毕竟每个博客站点的数据量并不是很大。在当时，单个数据库便于管理是个很好的理由。

网站最初建立的时期，这样的设计工作得很好，但很快就发展成大规模的数据库操作。现在他们在 300 台数据库服务器上存储 7 百万博客数据。每台服务器为一部分用户服务。

Barry 添加新的服务器时，对存储着所有用户博客信息的单一数据库进行拆分是非常痛苦的事情。通过将每个用户的数据分开存储到单独的数据库中，Barry 发现将一个用户的数据从一台服务器转移到另一台服务器变得异常简单。由于用户总是在不断地流动，有些用户的博客流量非常高，而另一些用户的博客则相对比较冷清，Barry 所负责的不断调整服务器之间的负载均衡工作就变得很重要。

备份并恢复一个中等规模的数据库比操作一个存储着 TB 级数据的数据库要方便得多。比如，如果一个用户打电话来说由于输入了错误的数据，他们的数据现在变得一团糟，Barry 该怎么恢复这个用户的数据，如果所有用户的数据都存于同一个巨大的数据库中？

尽管将数据对象模型化并将整个对象中的所有东西映射到一个单独的数据库中的做法没有错，但合理地将大小超过临界值的数据库拆分开能简化数据库管理的工作。

9.5 解决方案：分区及标准化

当一张表的数据量变得非常巨大时，除了手动拆分这张表，还有更好的办法来提升查询性能。这些方法就包括了水平分区、垂直分区以及使用关联表。

9.5.1 使用水平分区

你可以使用一种称为水平分区或者分片的数据库特性来分割大数据量的表，同时又不用担心那些分割表所带来的缺陷。你仅需要定义一些规则来拆分一张逻辑表，数据库会为你管理余下的所有事情。物理上来说，表的确是被拆分了，但你依旧可以像查询单一表那样执行 SQL 查询语句。

定义每个表拆分的方式是非常灵活的。比如，使用 MySQL5.1 所支持的分区特性，你可以在 CREATE TABLE 语句中将分区规则作为可选参数。

Metadata-Tribbles/soln/horiz-partition.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  -- other columns
  date_reported DATE
) PARTITION BY HASH ( YEAR(date_reported) )
  PARTITIONS 4;
```

上例中分割数据库的方式和这章最开始讲到的方法类似，根据 date_reported 列里的年份对数据进行拆分。然而，这么做的优势在于，相比于手动拆分表，你永远不用担心数据会放入错误的分割表中，即使 date_reported 列的值更新了。而且，你不必引用所有的分割表就能对 Bugs 表进行整体的查询操作。

实际存储数据的物理表在本例中被固定设置为 4 张。当记录的年份跨度超过 4 年，某一个分区将用来存储多于一年的数据。年份跨度不断增长，这样的现象也会不断重演。你不必添加新的分区，除非分区里的数据量变得非常巨大，让你觉得需要重新分区。

分区在 SQL 标准中并没有定义，因此每个不同的数据库实现这一功能的方式都是非标准的。对应的术语、语法和明确的特性定义在不同的数据库中有非常大的区别。然而，某些形式的分区如今已经被很大一部分数据库所支持了。

9.5.2 使用垂直分区

鉴于水平分区是根据行来对表进行拆分的，垂直分区就是根据列来对表进行拆分。当某些列非常庞大或者很少使用的时候，对表进行按列拆分会比较有优势。

BLOB 类型和 TEXT 类型的列的大小是可变的，可能非常大。为了提高存储和查询的性能，有

些数据库自动地将这些类型的列和表中其他的列分开进行存储。如果你进行一个不包含 BLOB 或者 TEXT 类型的查询，就可以更高效地获取其他的列。但如果使用一个通配符“*”来进行查询，数据库会返回这张表中所包含的所有列，包括那些类型为 BLOB 或者 TEXT 的列。

比如说，在缺陷数据库中，我们可能会在 Products 表中为每个单独的产品存储一份安装文件。这种文件都是自解压的，在 Windows 上的后缀通常为 .exe，在 Mac 上后缀为 .dmg。这种文件通常都很大，但 BLOB 类型的列可以存储庞大的二进制数据。

从逻辑上讲，安装文件是 Products 表的一个属性，但在绝大多数针对这张表的查询中，安装程序通常是不需要的。如果你有使用通配符“*”进行查询的习惯，那么将如此大的文件存储在 Products 表中，而且又不经常使用，很容易就会在查询时遗漏这一点，从而造成不必要的性能问题。

正确的做法是将 BLOB 列存在另一张表中，和 Products 表分离但又与其相关联。让这张 BLOB 表的主键同时作为一个指向 Products 表的外键，用以确保每个产品最多有一条与之对应的安装包记录。

Metadata-Tribbles/soln/vert-partition.sql

```
CREATE TABLE ProductInstallers (
  product_id      BIGINT UNSIGNED PRIMARY KEY,
  installer_image BLOB,
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

之前的例子虽然比较极端，但它确实展示了将一些列从某一张表中分离出来的优势。比如，在 MySQL 的 MyISAM 存储引擎中，对一个所有行的大小都是固定的表是最高效的。VARCHAR 是一个可变长数据类型，因此，只要表中出现一个这样类型的字段，那就无法享受固定长度的查询速度优势。如果你将所有可变长字段都存储在分离的表中，对主表的查询效率就能有所提高（即使只有一点点）。

Metadata-Tribbles/soln/separate-fixed-length.sql

```
CREATE TABLE Bugs (
  bug_id        SERIAL PRIMARY KEY, -- fixed length data type
  summary       CHAR(80),           -- fixed length data type
  date_reported DATE,               -- fixed length data type
  reported_by   BIGINT UNSIGNED,    -- fixed length data type
  FOREIGN KEY  (reported_by) REFERENCES Accounts(account_id)
);

CREATE TABLE BugDescriptions (
  bug_id       BIGINT UNSIGNED PRIMARY KEY,
  description  VARCHAR(1000),      -- variable length data type
  resolution   VARCHAR(1000)       -- variable length data type
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

9.5.3 解决元数据分裂列

和我们在第 8 章多列属性中使用的解决方案类似，解决元数据分裂的改进方案就是创建关联表。

Metadata-Tribbles/soln/create-history-table.sql

```
CREATE TABLE ProjectHistory (
  project_id  BIGINT,
  year        SMALLINT,
  bugs_fixed  INT,
  PRIMARY KEY (project_id, year),
  FOREIGN KEY (project_id) REFERENCES Projects(project_id)
);
```

使用每行一个项目、每一列记录一年的 Bug 修复数量，还不如使用多行、仅用一列记录修复的 Bug 数量。如果你这样定义表，就不需要为随后的年份增加新列，随着时间的增长，你可以为每个项目存储任意数量的记录。

别让数据繁衍元数据。

第 10 章
取整错误

10.0 乘以 0.1 未必就是 1.0。

▶ 布莱思 • 克尼汉

老板要求你根据每个程序员修复每个 Bug 所花费的时间，来计算并给出一个关于项目开发时间成本的报表。每个在 Accounts 表中的程序员都有不同的时薪，因此你统计出每个程序员花在修复每个 Bug 上的时间，并用其乘以对应的时薪。

Rounding-Errors/intro/cost-per-bug.sql

```
SELECT b.bug_id, b.hours * a.hourly_rate AS cost_per_bug
FROM Bugs AS b
  JOIN Accounts AS a ON (b.assigned_to = a.account_id);
```

要实现这样的查询功能，你需要在 Bugs 和 Accounts 表中创建新的列。这些新的列都应该支持浮点数，因为你需要精确地统计这些开销。你决定将这些列的类型定义为 FLOAT，因为这种类型支持浮点数。

Rounding-Errors/intro/float-columns.sql

```
ALTER TABLE Bugs ADD COLUMN hours FLOAT;

ALTER TABLE Accounts ADD COLUMN hourly_rate FLOAT;
```

通过分析与 Bug 相关的工作日志及程序员的时薪，你更新了这些新列，测试了相关数据，然后结束了当天的工作。

第二天，你的老板拿了份项目开销报表出现在你的办公室里。“这些数字不正确，”他咬牙切齿地说道：“我自己手动算过一次，而你的报告是错的——虽然很小，只差几美元。你怎么解释这个问题？”你开始冒冷汗，这样简单的计算哪里会出问题呢？

10.1 目标：使用小数取代整数

整型是一个很有用的数据类型，但只能存储整数，比如 1、327 或者-19 之类的。它不能表述

像 2.5 这样的浮点数。如果数据对精度要求很高，你需要使用另一种数据类型来取代整型。比如，算钱的时候通常需要精确到小数点后两位，像$19.95 这样。

因此，本章的目标就是存储非整数类型的数字，并且在基本运算中使用它们。还有另一个目标，运算的结果必须正确。当然，这个目标是最基本的要求，实现它是理所当然的。

10.2 反模式：使用 FLOAT 类型

大多数编程语言都支持实数类型，使用关键字 `float` 或者 `double`。SQL 也使用相同的关键字支持类似的数据类型。很多程序员很自然地就会在需要使用浮点数的地方使用 SQL `FLOAT` 类型，因为他们习惯于使用 `float` 类型编程。

SQL 中的 `FLOAT` 类型，就和其他大多数编程语言的 `float` 类型一样，根据 IEEE 754 标准使用二进制格式编码实数数据。你需要了解一些浮点数的定义规范，才能有效地使用这个数据类型。

10.2.1 舍入的必要性

很多程序员并不清楚浮点类型的特性：并不是所有在十进制中描述的信息都能使用二进制存储。出于一些必要的因素，浮点数通常会舍入到一个非常接近的值。

让我们更加直观地了解这个取整操作。举例来说，1/3 用一个无限循环的十进制可以表示为 0.333…，真实的值无法完整地写出来，因为需要写无限多个 3。小数点后数字的个数表示了这个数字的精确度，因此，无限循环地写下 3，能够无限接近于 1/3 的精确值。

折中的办法是限制精度，选择一个尽可能接近原始值的数字，比如 0.333。然而，这也意味着这个数字不是我们所希望的那个值。

$$\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1.000$$
$$0.333 + 0.333 + 0.333 = 0.999$$

即使提高了精度，仍旧不能将这三个近似值加起来得到 1.0。这是使用有限精度的数表示无限小数时的必要妥协。

$$\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1.000000$$
$$0.333333 + 0.333333 + 0.333333 = 0.999999$$

这意味着，你能想到的某些合理的数是无法用有限精度表示的。你可能觉得这样问题不大，因为你确实不可能输入一个无限小数，因而认为，既然实际输入的值是有限精度的，那么也应该能以相同的精度存储？但很不幸，事实并非如此。

IEEE 754 使用二进制表示浮点数。十进制中的无限小数在二进制中的表达方式是完全不同的。然而一些十进制有限小数，比如 59.95，在二进制中却需要表示为无限小数。`FLOAT` 类型无

法表达无限小数，因而，它存储了二进制表示中最接近 59.95 的值，用十进制表示等于 59.950000762939。

有些值碰巧在两种表达方式中能达到相同的精度，从理论上来说，如果你了解 IEEE 754 标准的细节，就可以预测一个给定的十进制数如何以二进制形式存储。但在实际中，大多数的人不会关心每一个他们正在使用的浮点数的实际精度。你无法保证一个 FLOAT 类型的列中存储的值都是符合精度需求的，因此你的程序应该认为这一列中的每个值都是经过舍入的。

有些数据库支持其他相类似的数据类型 DOUBLE PRECISION 和 REAL。包括 FLOAT 在内的三种数据类型的理论精度对于不同的数据库实现来说不尽相同，但它们都使用有限个二进制位来存储浮点数，因此它们都有着相似的取整行为。

10.2.2 在SQL中使用FLOAT

有些数据库能够通过某种方式弥补数据的不精确性，输出我们所期望的值。

IEEE 754 标准简介

浮点数二进制表达的规范的提案可以追溯到 1979 年，最终是在 1985 年定稿。如今已经广泛地被各种程序、编程语言及微处理器所实现。

这个规范使用三段编码：一段用来表示一个值的小数部分，一段用来表示其偏置指数，剩余的另外一位表示符号。在科学计算中，IEEE 754 标准的使用非常广泛，它同时能用来描述极大值以及极小值。这个标准不仅仅支持实数，其取值范围比固定大小的整型还要大。双精度浮点类型所支持的取值范围更大。因此，对于科学计算类的程序来说，这几个类型是非常有用的。

但是大多数使用浮点数的场合是用来算钱。对于钱的计算来说并不需要使用 IEEE 754 标准。因为本章所介绍的刻度化十进制格式能非常简单且精确地处理与钱相关的操作。

参考维基百科的文章（http://en.wikipedia.org/wiki/IEEE_754-1985）或者 David Goldberg 的文章“What Every Computer Scientist Should Know About Floating-Point Arithmetic”[Gol91]能更好地了解这个标准。

Goldberg 的文章可以查看http://www.validlab.com/goldberg/paper.pdf。

Rounding-Errors/anti/select-rate.sql

```
SELECT hourly_rate FROM Accounts WHERE account_id = 123;
```

Returns: 59.95

但 FLOAT 类型的列中实际存储的数据可能并不完全等于它的值。如果将这个值扩大十亿倍，就能看到其中的区别：

Rounding-Errors/anti/magnify-rate.sql

```
SELECT hourly_rate * 1000000000 FROM Accounts WHERE account_id = 123;
```

Returns: 59950000762.939

你可能希望上面那个扩大的查询返回的结果应该为 59950000000.000。这说明 IEEE 754 标准的二进制模式将 59.95 转化成了一个可以用有限精度表示的值。在这个例子中，取整后的值与原值的误差为千万分之一以内，对于大多数的运算来说已经足够精确了。

然而，对于某些运算来说这样的误差还是不可容忍的的。最简单的例子就是用 FLOAT 进行比较操作。

Rounding-Errors/anti/inexact.sql

```
SELECT * FROM Accounts WHERE hourly_rate = 59.95;
```

Result: empty set; no rows match.

我们知道在 hourly_rate 列中实际存储的值比 59.95 要稍微大一点点。即使你给 account_id 为 123 的 hourly_rate 赋值为 59.95，之前的那个查询也会以失败而告终。

通常的变通方案是将浮点数视作“近似相等”，即两个值之间的差值足够小就认为它们相等。我们将两个值相减，并使用 SQL 中的 ABS() 函数取其绝对值。如果结果为 0，则表示两个数绝对相等。如果结果足够小，那我们就认为这两个数是近似相等的。下面的这个查询能够返回正确的结果：

Rounding-Errors/anti/threshold.sql

```
SELECT * FROM Accounts WHERE ABS(hourly_rate - 59.95) < 0.000001;
```

然而，即使是如此细小的差值在对精度要求较高的比较中也会失败：

Rounding-Errors/anti/threshold.sql

```
SELECT * FROM Accounts WHERE ABS(hourly_rate - 59.95) < 0.0000001;
```

由于十进制数和取整后的二进制数的绝对值不相同，我们需要合适的阈值。

另一个由于使用非精确的 FLOAT 造成误差的情况，是使用合计函数计算很多值的时候。比如，如果使用 SUM() 函数计算一列中的所有值，那最终得到的总和会受到这一列中所有非精确值的影响。

Rounding-Errors/anti/cumulative.sql

```
SELECT SUM( b.hours * a.hourly_rate ) AS project_cost
FROM Bugs AS b
JOIN Accounts AS a ON (b.assigned_to = a.account_id);
```

非精确浮点数所积累的影响对于求和之外的合计运算来说会更大。虽然误差看起来非常小，但其累加起来的效果不可忽视。比如，如果你用 1 精确地乘以 1.0，那无论执行多少次，结果总是 1。然而，如果这个乘数因子实际上是 0.999，结果就完全不同。如果你用 1 乘以 0.999 一千次，你得到的结果将约等于 0.3677。这样的操作执行次数越多，误差就越大。

实际中需要连续多次进行浮点数乘法运算的例子，是在金融项目中计算复利。使用非精确浮点数所造成的误差看起来很小，但会不断累加。因此，在金融项目中使用精确值是非常重要的。

10.3 如何识别反模式

实际上，任何使用 FLOAT、REAL 或者 DOUBLE PRECISION 类型的设计都有可能是反模式。大多数应用程序使用的浮点数的取值范围并不需要达到 IEEE 754 标准所定义的最大/最小区间。

似乎在 SQL 中使用 FLOAT 类型是很自然的事情，毕竟它和大多数编程语言中的浮点类型所使用的关键字是一样的。但对于浮点数的存储，我们还有更好的选择。

10.4 合理使用反模式

当你需要存储的数据的取值范围很大，大于 INTEGER 和 NUMERIC 这两个类型所支持的范围时，FLOAT 就是你的选择。科学计算类的程序就是 FLOAT 通常的应用场合。

Oracle 使用 FLOAT 类型表示的是一个精确值，而 BINARY_FLOAT 类型是一个非精确值，使用的是 IEEE 754 标准编码。

10.5 解决方案：使用 NUMERIC 类型

使用 SQL 中的 NUMERIC 或 DECIMAL 类型来代替 FLOAT 及与其类似的数据类型进行固定精度的小数存储。

Rounding-Errors/soln/numeric-columns.sql

```
ALTER TABLE Bugs ADD COLUMN hours NUMERIC(9,2);

ALTER TABLE Accounts ADD COLUMN hourly_rate NUMERIC(9,2);
```

这些数据类型精确地根据你定义这一列时指定的精度来存储数据。通过类似于指定 VARCHAR 的长度的方式，将精度作为类型参数来定义列的类型。其精度所指的是，在这一列中的每个值最多所能包含的有效数字的个数。精度为 9 意味着你可以存储 123456789，而 1234567890[①]则为非法值。

① 在一些厂商的数据库中，列的大小内部取整到最接近的字节、字或是双字，所以 NUMERIC 列的最大值可能有比你规定的要多的数位。

你也可以使用该类型的第二参数来指定其刻度。这里的刻度即指小数点后的位数。小数部分的数字也算在其有效位中，因此，精度 9 刻度 2 意味着可以存储 1234567.89，而 12345678.91 或者 123456.789 都是非法值。

应用在某一列上的精度和刻度会影响到这一列中的每一行记录，也就是说，你无法在某些行上存储刻度为 2 的记录，同时又在另一些行上存储刻度为 4 的记录。在 SQL 中一列的数据类型对于这一列中的每条记录都是通用的（就如同定义了 VARCHAR(20)意味着每一行都只能存储最大长度为 20 的字符串）。

NUMERIC 和 DECIMAL 的优势之处在于，它们不会像 FLOAT 类型那样对存储的有理数进行舍入操作。假设你输入 59.95，就可以确信实际存储的数据就是 59.95。当使用存储的数据与原始数据 59.95 进行比较时，必然返回两者相等。

Rounding-Errors/soln/exact.sql

```
SELECT hourly_rate FROM Accounts WHERE hourly_rate = 59.95;
```

Returns: 59.95

同样地，如果你按比例将值扩展十亿倍，就可以得到所期望的值：

Rounding-Errors/soln/magnify-rate-exact.sql

```
SELECT hourly_rate * 1000000000 FROM Accounts WHERE hourly_rate = 59.95;
```

Returns: 59950000000

NUMERIC 和 DECIMAL 这两个类型的行为是一样的，两者没有任何区别。DEC 也是 DECIMAL 的简称。

你仍旧无法存储无限精度的数据，诸如 1/3，但至少我们对十进制数的这些约束有了更深的了解。

如果你需要精确地表示十进制数，使用 NUMERIC 类型。FLOAT 类型无法表示很多十进制的有理数，因此它们应该当成非精确值来处理。

尽可能不要使用浮点数。

第 11 章 每日新花样

当变量有限且可以一一列举，当变量间的组合是不重复且清晰的，那就存在科学。

▶ 保罗·瓦勒里

在个人信息表中，称呼（salutation）列可以只有有限的几个候选值。基本上你只需要支持 Mr.、Mrs.、Ms.、Dr.以及 Rev.，这些称谓几乎覆盖了所有人。你可以在声明这个列的时候使用数据类型或者约束来指定这些候选值，因此不会有人往 salutation 列中加入无效值。

31-Flavors/intro/create-table.sql

```
CREATE TABLE PersonalContacts (
  -- other columns
  salutation VARCHAR(4)
    CHECK (salutation IN ('Mr.', 'Mrs.', 'Ms.', 'Dr.', 'Rev.')),
);
```

这一列基本上可以保持稳定，因为你不需要支持别的称呼了，是这样的吗？

遗憾的是，你的老板告诉你公司正准备在法国创建一家分公司。你需要支持 M.、Mme.以及 Mlle.这些称呼。你的任务就是让你的联系人表能接受这些值。这是一项很精细的任务，不中断表的读写操作可能无法完成这一任务。

你同时还得考虑，你的老板提到了下月可能会在巴西建立办事处的事情。

11.1 目标：限定列的有效值

将一列的有效字段值约束在一个固定的集合内是非常有用处的。如果我们可以确保一列中永远不会包含无效字段，那对于使用方来说，逻辑会变得非常简单。

比如在 Bugs 表中，status 列标记了一个给定的 Bug 是 NEW、IN PROGRESS、FIXED 等状态。这些值的意义与在项目中如何管理这些 Bug 有关，但我们现在所关心的是 status 这一列中的值必须限定在所列出来的这几个值中。

理想情况下，我们希望数据库能够拒绝无效值的输入：

31-Flavors/obj/insert-invalid.sql

```
INSERT INTO Bugs (status) VALUES ('NEW'); -- OK

INSERT INTO Bugs (status) VALUES ('BANANA'); -- Error!
```

芭斯罗缤的 31 种冰激凌

从 1953 年开始，著名的冰激凌连锁店芭斯罗缤（Baskin-Robbins）在一个月中每天提供一种不同的口味。他们使用“31 Flavors”这个口号很多年。

如今，60 多年后，芭斯罗缤提供 21 种经典口味，12 种季节性口味，16 种地区性口味，另外还有各种各样的 Bright Choices 和 Flavors of the Month。以前芭斯罗缤的冰激凌口味根据他们的招牌是固定不变的，但他们后来对此做了扩展，可配置，可变化。在你设计数据库时，也可以使用同样的方式——实际上，你确实应该使用这种方式。

11.2 反模式：在列定义上指定可选值

很多数据库设计人员趋向于在定义列的时候指定所有可选的有效数据。列的定义是元数据的一部分，也就是表结构定义的一部分。

比如说，你可以对某一列定义一个检查约束项。这个约束不允许往列中插入或者更新任何会导致约束失败的值。

31-Flavors/anti/create-table-check.sql

```
CREATE TABLE Bugs (
  -- other columns
  status   VARCHAR(20) CHECK (status IN ('NEW', 'IN PROGRESS', 'FIXED'))
);
```

MySQL 支持使用 ENUM 关键字来约束一列的取值范围，但这并不是一个标准类型。

31-Flavors/anti/create-table-enum.sql

```
CREATE TABLE Bugs (
  -- other columns
  status ENUM('NEW', 'IN PROGRESS', 'FIXED'),
);
```

在 MySQL 的实现中，即使你使用字符串表示 ENUM 里的值，但实际存储在列中的数据是这些值在定义时的序数。因此，这列的数据是字节对齐的，当你进行一次排序查询时，结果是按照实际存储的序号进行排序的，而非对应字符串的字母序。这可能并不是你所希望的。

其他的解决方案包含了域以及用户自定义类型（UDT）。你可以使用这些方法来约束某一列

只能接受一个特定集合中的数据，并且能很方便地将这个约束应用到整个域上。但这些特性并没有得到大多数关系数据库系统的支持。

最后一个办法，你还可以编写一个触发器，当修改指定列的内容时触发，将被修改的值和允许输入的值进行匹配，如果不符合则产生一个错误中断操作。

所有的这些解决方案都有一定的缺陷。下面就来一一描述这些问题。

11.2.1　中间的是哪个

假设你正在为 Bug 跟踪服务开发一个用户界面程序，它允许用户编辑 Bug 报告。为了让界面能够引导用户选择一个合适的 status 值，你选择使用一个包含所有可选值的下拉菜单。你要如何查询数据库来获取当前可以输入到 status 列中值的枚举列表呢？

你的第一反应可能是查询当前列中所有正在被使用的值，使用如下这个简单的查询：

31-Flavors/anti/distinct.sql

```
SELECT DISTINCT status FROM Bugs;
```

然而，如果所有的 Bug 都是新建的，上面这个查询只会返回 NEW。如果你使用这样的结果集来填充用户界面中的控件，这就变成了一个先有鸡还是先有蛋的问题：你无法将一个 Bug 的状态改变为当前使用状态以外的值。

要获得所有允许输入的 status 候选值，你需要查询这一列的元数据。大多数 SQL 数据库支持使用系统视图来完成这种查询需求，但是使用起来是很复杂的。比如，如果你使用 MySQL 的 ENUM 类型，可以使用如下的查询语句来查询 INFORMATION_SCHEMA 系统视图：

31-Flavors/anti/information-schema.sql

```
SELECT column_type
FROM information_schema.columns
WHERE table_schema = 'bugtracker_schema'
  AND table_name = 'bugs'
  AND column_name = 'status';
```

你无法简单地从 INFORMATION_SCHEMA 的结果集中获取单独的枚举值，而是得到了一个包含 Check 约束或者 ENUM 类型声明的字符串。比如，在 MySQL 中，上述查询就会返回一个类型为 LONGTEXT、内容为 ENUM('NEW', 'IN PROGRESS', 'FIXED')的结果，结果中包含了括号、逗号及单引号。你必须额外编写一段程序代码来解析这个字符串，将每个引号对中的数据独立抽取出来，才能在控件中使用。

对于获取 Check 约束、域或者 UDT 信息的查询来说，过程会更加复杂。大多数开发人员非常勇敢地在程序中手动维护这样一个列表。当程序数据和数据库的元数据不同步时，程序很容易就崩溃了。

11.2.2 添加新口味

最常见的改变就是添加或删除一个候选值。没有什么语法支持从 ENUM 或者 Check 约束中添加或删除一个值，你只能使用一个新的集合重新定义这一列。如下是一个更新 MySQL ENUM 的例子，目的是将 DUPLICATE 添加到 status 的候选值列表中：

31-Flavors/anti/add-enum-value.sql

```
ALTER TABLE Bugs MODIFY COLUMN status
  ENUM('NEW', 'IN PROGRESS', 'FIXED', 'DUPLICATE');
```

你先要知道之前的定义允许 NEW、IN PROGRESS 和 FIXED，这样问题又绕回到了之前如何获取这些值上了。

一些数据库要求只有表为空表时才能改变某一列的定义。你可能需要先转存这张表中的数据，清空这张表，重新定义它，然后再将数据重新导入，这个过程会使得这张表处于无法访问的状态。这种工作非常常见，以致它已经有了个名字：ETL，表示“抽取、转换和加载”。有一些数据库支持使用 ALTER TABLE 指令重新构建一张使用中的表，但这个操作依旧是非常复杂的，并且其开销也很巨大。

作为一个策略问题，修改元数据——意味着修改表或列的定义——应该是极少的，并且需要大量的测试以保证质量。如果你需要修改元数据来对一个 ENUM 定义进行添加或删除操作，就不得不投入大量的测试时间或者让很多工程师关注这个修改所带来的影响。否则，这样的修改就会导致额外的风险，让你的程序变得不稳定。

11.2.3 老的口味永不消失

如果你打算废弃一个选项，你可能会为老数据而烦恼。比如，将质量控制流程中的 FIXED 状态拆分成 CODE COMPLETE 和 VERIFIED 两个状态：

31-Flavors/anti/remove-enum-value.sql

```
ALTER TABLE Bugs MODIFY COLUMN status
  ENUM('NEW', 'IN PROGRESS', 'CODE COMPLETE', 'VERIFIED');
```

如果移除 FIXED，你要如何处理已经是 FIXED 状态的数据？将所有的 FIXED 状态修改为 VERIFIED？还是将其设为空或者默认值？

你可能不得不保留这个老数据已使用的废弃选项。但又要如何在用户界面上区分废弃的和可用的 Bug 状态候选值呢？

11.2.4 可移植性低下

Check 约束、域和 UDT 在各种数据库中的支持形式并不统一。ENUM 是 MySQL 独有的特性。

每一个数据库对于列的候选值列表长度的定义都不尽相同。触发器的语法也区别很大。这些差异使得你在需要支持多种数据库时很难选择一个合适的解决方案。

11.3 如何识别反模式

使用ENUM或者Check约束时遇到的问题可能是候选值集合并不固定。如果你正考虑使用ENUM，首先问一下自己，候选值的集合是否需要改变或者是否可能改变。如果答案为“是”，那使用ENUM就不见得是好主意。

- “我们不得不将数据库下线，才能在程序菜单中加入一个新的选项。如果一切顺利，整个过程将不超过 3 小时。”
  这说明候选值的集合是直接写入列的定义中的。然而，完成这样的升级操作理应不停止服务。
- “这个 status 列可以填入这些候选值中的一个。我们不应该改变这个候选值列表。”
  “不应该”是一个很模棱两可的说法，它和“不能”是完全不同的意思。
- “程序代码中关于业务规则的选项列表和数据库中的值又不同步了！”
  这就是在两个地方维护同一套数据的风险。

11.4 合理使用反模式

就像我们讨论的那样，ENUM 在候选值几乎不变的情况下所造成的问题很少。通过查询获取元数据依旧是很麻烦的，但是你可以在程序代码中维护一份列表而不用担心不同步的问题。

ENUM 在存储没有业务逻辑且不需要改变的候选值时是非常方便的。比如存储一对二选一且相互对立的值：LEFT/RIGHT、ACTIVE/INACTIVE、ON/OFF、INTERNAL/EXTERNAL 等。

Check 约束可以在更多的场景下使用，不仅仅是实现一个类 ENUM 的机制，比如用来检查一个时间区间中 start 永远小于 end。

11.5 解决方案：在数据中指定值

有一个更好的解决方案来约束一列中的可选值：创建一张*检查表*，每一行包含一个允许在 Bugs.status 列中出现的候选值；然后定义一个外键约束，让 Bugs.status 引用这个新表。

31-Flavors/soln/create-lookup-table.sql

```
CREATE TABLE BugStatus (
  status  VARCHAR(20) PRIMARY KEY
);

INSERT INTO BugStatus (status) VALUES ('NEW'), ('IN PROGRESS'), ('FIXED');
```

```
CREATE TABLE Bugs (
  -- other columns
  status  VARCHAR(20),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
    ON UPDATE CASCADE
);
```

当你插入或更新 Bugs 表中的一条记录时，必须使用存在于 BugStatus 表中的一个 status 值。这样做类似于 ENUM 和 Check 约束一样，能够确保 status 值的有效性；同时，这样的方案还在其他地方体现出了它的灵活性。

11.5.1　查询候选值集合

候选值集合现在是存储在数据表中，而不是像 ENUM 那样存储在元数据中。你可以使用 SELECT 对这张检查表进行查询来获取相关数据，和查询别的表没什么区别。这使得获取候选值集合并作为数据集在用户界面中展示变得非常简单。你甚至可以对用户可选值进行排序操作。

31-Flavors/soln/query-canonical-values.sql

```
SELECT status FROM BugStatus ORDER by status;
```

11.5.2　更新检查表中的值

当你使用检查表时，可以使用原始的 INSERT 语句向其中加入一个值。你可以不中断对表的访问就完成这样的改变。你也不需要重新定义任何列，不需要安排下线时间，或者执行一次 ETL 操作。同样，你也不需要在执行添加或删除操作前知道当前检查表中有哪些值。

31-Flavors/soln/insert-value.sql

```
INSERT INTO BugStatus (status) VALUES ('DUPLICATE');
```

如果定义外键时使用了 ON UPDATE CASCADE 选项，重命名一个值也会变得非常方便。

31-Flavors/soln/update-value.sql

```
UPDATE BugStatus SET status = 'INVALID' WHERE status = 'BOGUS';
```

11.5.3　支持废弃数据

如果检查表中的一个值被 Bugs 表中的数据引用了，那就不能删除它了。status 列上的外键确保了引用完整性，因此，status 列引用的值必须存在于检查表中。

然而，你可以在检查表中增加另一个属性列来标记一些废弃数据。这样做允许你保留 Bugs.status 列中的历史数据，同时又能够区分哪些值是能够出现在用户界面上的。

31-Flavors/soln/inactive.sql

```
ALTER TABLE BugStatus ADD COLUMN active
  ENUM('INACTIVE', 'ACTIVE') NOT NULL DEFAULT 'ACTIVE';
```

使用 UPDATE 代替 DELETE 来废弃一个值：

31-Flavors/soln/update-inactive.sql

```
UPDATE BugStatus SET active = 'INACTIVE' WHERE status = 'DUPLICATE';
```

当要获取在界面上展示的候选值列表时，在查询条件中增加一个 status 为 ACTIVE 的约束：

31-Flavors/soln/select-active.sql

```
SELECT status FROM BugStatus WHERE active = 'ACTIVE';
```

这样的解决方案相比于 ENUM 或者 Check 约束来说更加灵活，因为前两者无法为每个值提供额外属性。

11.5.4　良好的可移植性

不同于 ENUM 类型、Check 约束，或者域及 UDT，检查表的解决方案只依赖于最基本的 SQL 特性：使用外键确保引用完整性。这使得该解决方案的兼容性得到了保证。

由于在每一行中存储一个候选值，就使得检查表在理论上可以支持无限多个候选值。

在验证固定集合的候选值时使用元数据。

在验证可变集合的候选值时使用数据。

第12章 幽灵文件

> 当一个理论看上去像是唯一可能的理论时，那意味着你既不理解这个理论，也不理解它所要解决的问题。
>
> ► 卡尔·波普尔

你的数据库服务器在大灾难中没有幸存下来，在清理时发现硬盘机架整个倾倒了，并且摔坏了。幸运的是，没有人因此而受伤，但是大量的硬盘因此而损坏，甚至存放机架的楼层在机架倾倒时被砸穿了。所幸，IT 部门的灾备做得比较好：他们每天都为每个重要的系统做了备份，并且快速地在新的服务器上部署了新的服务，恢复了数据库。

冒烟测试进行没多久后发现了一个问题：你的程序将图像与很多数据库字段进行了关联，但是所有这些图片都不见了！你立刻打给了 IT 技术部门。

“我们恢复了数据库并且验证了这是和上次备份一样的完整副本，”这个技术人员说：“图片是存在哪里的？”

你现在想起来了，在这个应用中，图片是存在数据库之外的，普通文件都是存在文件系统中的。数据库里只存了图片的路径，通过程序去打开对应路径的图片。“图片是以文件形式存储的。它们是在`/var`目录下，和数据库一起。”

这个技术人员摇了摇头。“除非你做过特殊说明，否则我们不备份`/var`下的内容。当然我们会备份数据库的文件，但`/var`目录经常是存放日志、缓存或其他临时文件的地方。默认情况下，这些数据都是不备份的。”

你心痛啊。那个目录下存了超过 11 000 张在产品分类数据库中使用的图片。大多数可能在其他地方还有备份，但要把它们都放到一起，重新编排，并且为网络搜索重做缩略图要花好几个星期啊！

12.1 目标：存储图片或其他多媒体大文件

如今，图片等多媒体文件已经广泛使用在很多程序中。有时，多媒体文件和数据库中的一些

实体关联。比如，你可能会允许一个用户在提交评论时显示一个头像。在我们的 Bug 数据库中，Bug 通常需要附带一个截屏来展示具体情况。

本章的目标就是要存储这些图片并且将其和数据库实体（诸如用户账户或者 Bug）关联起来。当查询这些实体时，我们需要确保同时能获取与其关联的图片。

12.2 反模式：假设你必须使用文件系统

理论上来说，图片是一张表中的一个字段，在 Accounts 表中可能会有一个 portrait_image 列。

Phantom-Files/anti/create-accounts.sql

```
CREATE TABLE Accounts (
  account_id      SERIAL PRIMARY KEY
  account_name    VARCHAR(20),
  portrait_image  BLOB
);
```

同样地，你可以在从属表中存储多张同类型的图片。比如，每个 Bug 都可以有多张屏幕截图。

Phantom-Files/anti/create-screenshots.sql

```
CREATE TABLE Screenshots (
  bug_id            BIGINT UNSIGNED NOT NULL,
  image_id          SERIAL NOT NULL,
  screenshot_image  BLOB,
  caption           VARCHAR(100),
  PRIMARY KEY       (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

这些都不难理解，但真正的重点问题是选择什么样的数据类型来存储图片？原始图片文件可以以二进制格式存储在 BLOB 类型中，就像之前我们存储超长字段那样。然而，很多人选择将图片存储在文件系统中，然后在数据库里用 VARCHAR 类型来记录对应的路径。

Phantom-Files/anti/create-screenshots-path.sql

```
CREATE TABLE Screenshots (
  bug_id            BIGINT UNSIGNED NOT NULL,
  image_id          BIGINT UNSIGNED NOT NULL,
  screenshot_path   VARCHAR(100),
  caption           VARCHAR(100),
  PRIMARY KEY       (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

开发人员激烈地争论着这个问题。两种方案都有很好的立足点，但是对于程序员来说通常只有一种选择，就是我们应该将文件存在数据库之外。可能我的观点有点不合时宜，但我依旧要在接下来的几节中指明这样的设计所面临的很现实的风险。

12.2.1 文件不支持DELETE

第一个问题就是垃圾回收。如果图片在数据库之外，并且你想要删除包含这个路径的记录，没有什么办法能自动地将路径对应的文件移除。

Phantom-Files/anti/delete.sql

```
DELETE FROM Screenshots WHERE bug_id = 1234 and image_id = 1;
```

除非你的应用程序设计成在删除记录的同时删除这些“无人领养”的图片，不然这些图片就会堆积在那里。

12.2.2 文件不支持事务隔离

通常，当更新或删除数据时，在使用 COMMIT 指令完成事务操作之前，所有的改变都对其他的客户端不可见。

然而，任何对数据库之外的文件的操作并非如此。如果你删除了一个文件，对于其他的客户端来说就立刻无法访问该图片了。并且如果你改变了文件的内容，其他的客户端可以立刻看到这些变更，而不是看到在事务未提交之前的文件状态。

Phantom-Files/anti/transaction.php

```
<?php

$stmt = $pdo->query("DELETE FROM Screenshots
                     WHERE bug_id = 1234 AND image_id =1");

unlink('images/screenshot1234-1.jpg');
// Other clients still see the row in the database,
// but not the image file.

$pdo->commit();
```

在实际生产过程中，这样的特例可能并不常见。同样，本例的影响也比较小；在 Web 程序中，丢失图片的情况很少见。但在别的情况下，后果就可能非常严重。

12.2.3 文件不支持回滚操作

出错情况下，或者程序逻辑要求取消变更时，对数据进行回滚操作是很平常的事情。

比如，你最开始执行了一句 DELETE 语句来删除一条记录并同时移除了对应的截屏文件，然后你回滚了这个操作，被删除的数据回来了，但文件已经没了。

Phantom-Files/anti/rollback.php

```
<?php
```

```
$stmt = $pdo->query("DELETE FROM Screenshots
                     WHERE bug_id = 1234 AND image_id =1");

unlink("images/screenshot1234-1.jpg");

$pdo->rollback();
```

数据库中的记录能够恢复，但文件不能。

12.2.4 文件不支持数据库备份工具

大多数数据库产品都提供客户端工具来协助备份使用中的数据库。比如 MySQL 提供了一个叫做 mysqldump 的组件，Oracle 提供了 rman，PostgreSQL 提供了 pg_dump，SQLite 提供了 .dump 命令等。使用备份工具非常重要，如果同一时刻别的客户端正在进行变更操作，将可能使你的备份包含不完整的变更，造成潜在的引用不完整性，甚至使得整个备份被破坏以至于无法用于恢复数据库。

但是备份工具并不知道如何将通过路径引用的那些文件也包含在备份操作当中。因此在备份一个数据库时，你需要记住执行一个两步操作：使用数据库备份工具，然后使用文件系统备份工具来收集外部图像文件。

即使在备份时包含了外部文件，也很难保证这些文件备份和你执行备份数据库的事务是同步的。程序可能在任何时间对图片文件做变更，也许就在你开始备份数据库之后不久。

12.2.5 文件不支持SQL的访问权限设置

外部文件会绕开通过 GRANT 和 REVOKE SQL 语句设定的访问权限。SQL 权限管理着对表和列的访问，但它们并不能应用到外部文件。

12.2.6 文件不是SQL数据类型

在 screenshot_path 字段中存储的路径就是一个字符串。数据库并不会验证这个字符串是一个有效的路径，也不会验证对应的文件是否存在。如果这个文件被重命名、移动或者删除了，数据库并不会自动更新对应的路径。任何将这个字符串作为路径处理的逻辑都依赖于你的程序逻辑。

Phantom-Files/anti/file-get.php

```
<?php

define('DATA_DIRECTORY', '/var/bugtracker/data/');

$stmt = $pdo->query("SELECT image_path FROM Screenshots
```

```
                    WHERE bug_id = 1234 AND image_id = 1");
$row = $stmt->fetch();
$image_path = $row[0];

// Read the actual image -- I hope the path is correct!
$image = file_get_contents(DATA_DIRECTORY . $image_path);
```

使用数据库的好处之一在于它能帮助我们保持数据完整性。当你将数据放在外部文件中时，就抛弃了数据库提供的这个好处，并且你不得不写更多的程序代码来执行本该由数据库进行的检查操作。

12.3 如何识别反模式

要发现这个反模式需要一定的调查。如果一个项目有指导软件管理员的文档，或者你有机会与设计项目的程序员（或者就是你自己）面谈，考虑一下如下几个问题的答案。

- 数据备份和恢复的过程是怎样的？怎么对一个备份进行验证？你有没有在一个干净的系统或者在别的系统上对备份恢复的数据进行测试？
- 图片文件堆积在那里，还是当它们孤立的时候就从系统中移除？移除它们的过程是怎样的？这是一个自动的还是手动的过程？
- 系统中的哪些用户有权限查看这些图片？进入权限是怎么限制的？当用户请求看他们无权查看的图片时会发生什么？
- 我能撤销对图片的变更吗？如果能，是应用程序来负责恢复图片之前的状态吗？

典型的使用反模式的项目通常没有考虑以上的几个或者全部的问题。并不是每个程序都需要有针对图片的很强的事务管理或者 SQL 访问控制。可能在备份过程中将数据库下线也是一个很好的方案。如果以上这些问题的回答不明确或者并没有立刻给出回答，那可以假设这个项目对于外部文件使用的并没有经过精心设计。

12.4 合理使用反模式

如下是一些将图片或者大文件存储在数据库之外的好理由。

- 这个数据库在没有图片的时候能精益很多，因为图片相比于简单的数据类型（比如整形和字符串）来说更大。
- 当不包含图片时备份数据库会更快并且备份的文件更小。你必须额外执行一次文件备份，但这比备份一个大型数据库要更容易管理。
- 如果图片是存储在数据库之外的文件系统里，对图片的预览或者编辑就能够使用更简单直接的处理方式。比如，如果你需要执行一个批处理编辑所有的图片，将其保存在数据库之外就是一个特别好的选择。

如果这些将图片存在文件系统中的好处是重要的，并且之前几节所描述的那些事项并不会破坏你的系统，那就可以肯定将图片存在数据库之外对于你的项目来说是正确的决定。

一些数据库产品提供了一些特殊的 SQL 数据类型，为支持对外部文件引用提供或多或少的透明性。Oracel 称这种类型为 BFILE，SQL Server 2008 称之为 FILESTREAM。

不要排除任何设计

我在 1992 年的时候为一个外包工程设计了一个将图片存储在数据库之外的程序。我的雇主承接了一个技术会议的注册程序。在会议即将开始之前，我们用一个照相机对每个与会人士拍照，并将他们的照片加到他们的注册信息中，同时也打印在他们的通行证上。

我的程序非常简单。每个图片都只能被一个客户端插入或者更新（如果这个人在拍照时眨眼了，或者不喜欢他们的照片，我们会在注册时就更换它）。没有复杂的事务处理的需求，也没有多客户端的并发访问或者回滚需求。我们甚至没有使用 SQL 的权限控制。预览图片的逻辑简单到根本不需要将其从数据库中提取出来。

我开发这个项目的时候，数据库及客户端技术还不像现在这么发达。于是我们有很充分的理由在这种情况下将图片直接存在文件系统目录中，并且使用应用层代码来维护。

你需要规划你的程序如何使用这些文件，并且了解本章所描述的反模式是否会影响到你的系统。做一个明智的决定，而不仅仅听那些将图片存在数据库之外的程序员的泛泛之谈。

12.5 解决方案：在需要时使用 BLOB 类型

如果在12.2节中所描述的任何问题适用于你的项目，你应该要考虑将图片从数据库之外转移到数据库之内。所有的数据库产品都支持 BLOB 类型，支持你存储任何二进制数据。

Phantom-Files/soln/create-screenshots.sql

```
CREATE TABLE Screenshots (
  bug_id            BIGINT UNSIGNED NOT NULL,
  image_id          BIGINT UNSIGNED NOT NULL,
  screenshot_image  BLOB,
  caption           VARCHAR(100),
  PRIMARY KEY       (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

如果你将图片存在一个 BLOB 类型的列中，所有的问题都将得到解决。

- 图片数据存储在数据库中。不需要额外的步骤加载它，也就没有“文件路径不正确”这样的风险。
- 删除一条记录同时也自动地删除了图片。

- 直到你提交了事务，否则对图片的改变是对其他客户端不可见的。
- 回滚事务会恢复图片之前的状态。
- 更新记录的时候会加锁，因此不会有别的客户端并发更新图片。
- 数据库备份会包含所有的图片。
- SQL 权限控制对图片也有效。

BLOB 类型的最大值依据不同的数据库产品而不同，但对于存储大部分的图片文件来说都是足够的。所有的数据库都支持 BLOB 或者类似的类型。比如，MySQL 提供了一个叫做 MEDIUMBLOB 的类型，支持最大 16MB 的数据，对于绝大部分图片来说都足够了。Oracle 提供了一个 LONG RAW 或者 BLOB 的类型，最大支持 2GB 或者 4GB 的长度。类似的类型在其他数据库产品中也能找到。

图片文件最开始都是以文件形式存储的，因此你需要一些途径将它们载入 BLOB 列。一些数据库产品提供了加载外部文件的函数。比如，MySQL 有个函数叫做 LOAD_FILE()，你可以用它来读取一个文件，然后将内容存到 BLOB 列中。

Phantom-Files/soln/load-file.sql

```sql
UPDATE Screenshots
SET screenshot_image = LOAD_FILE('images/screenshot1234-1.jpg')
WHERE bug_id = 1234 AND image_id = 1;
```

同样地，你也可以将 BLOB 列的内容存储到一个文件中去。比如，MySQL 有一个可选的 SELECT 子句能将一个查询完整地不加修改地存储到文件中去。

Phantom-Files/soln/dumpfile.sql

```sql
SELECT screenshot_image
INTO DUMPFILE 'images/screenshot1234-1.jpg'
FROM Screenshots
WHERE bug_id = 1234 AND image_id =1;
```

你也能够直接从 BLOB 字段中提取图片并且输出。在 Web 程序中，你可以将二进制数据以图片输出，但你需要将内容类型设置为合适的值。

Phantom-Files/soln/binary-content.php

```php
<?php

header('Content-type: image/jpg');

$stmt = $pdo->query("SELECT screenshot_image FROM Screenshots
                     WHERE bug_id = 1234 AND image_id = 1");
$row = $stmt->fetch();

print $row[0];
```

存储在数据库之外的数据不由数据库管理。

第13章 乱用索引

无论何时，在机器的帮助下找到结果，另一个问题就冒出来了——通过何种计算过程，能让机器最快地求出结果？

► 查尔斯 · 巴贝奇，《哲学家的生命历程》(1864)

“嗨！你有没有时间？我需要你的帮助。”带有俄克拉何马口音的人在电话那头对着你喊，他的声音也顺着数据中心的通风管道传过来。这是你们公司的数据库管理员的头儿。

“当然。”你有点心虚地回答道。他想干嘛？

“是这样，有一个你们的数据库服务在运行，它占了太多资源。”这个 DBA 继续说道，“我进去看了一下，然后发现了问题。在有些表上完全没有使用索引，而在其他表上索引又多得用不了。

“我们必须解决这个问题，或者给你们一台独立服务器，因为这台机器上的其他服务都无法正常使用了！”

“我很抱歉。事实上，我对数据库了解得不多。”你小心地回答，尽量让这个 DBA 冷静下来，“我们已经尽了最大的努力来猜测哪些是可以优化的，但显然这些工作只有你们这些专家才能做得好。有没有什么你能做的调整？”

“孩子，我做了所有能做的调整，这就是为什么到目前为止它还没有挂掉，”这个 DBA 回答道，“留给我的唯一选择是限制你们的程序对数据库的访问量，但我认为你们并不想要这样。我们必须停止猜测，首先我们要弄明白你们的程序到底需要数据库做什么。”

你觉得头都大了，小心谨慎地问道：“你有什么想法吗？坦白地说，我们组里并没有数据库方面的专家。”

“这不成问题。”这个 DBA 笑道，“你们确实对你们的程序了如指掌，对吧？而这部分是关键

内容，不过我可帮不上忙。我会派一个手下到你们那去帮忙安装一些正确的工具，然后我们将找到并消除你们的瓶颈。你们只需要一点点的指导。你很快就会明白的。”

13.1 目标：优化性能

性能是我从那些数据库开发人员那里听到的一个最普遍的问题。只要看一下任何技术会议的议题你即可明白：到处都充斥着工具和技术来让数据库再多工作一点。当我要做的讲座涉及如何规划数据库或写出更为可靠、安全、正确的 SQL 方面内容时，有可能他们唯一的疑问在于：“好吧，但是这些对性能有什么影响？”而我对此并不感到吃惊。

改善性能最好的技术就是在数据库中合理地使用索引。索引也是数据结构，它能使数据库将指定列中某个值快速定位在相应的行。索引提供了一种简单而高效的途径让数据库能够快速地找到需要的值，而不是野蛮地进行一次自上而下的全表遍历。

软件开发人员通常并不理解如何或何时使用索引。关于何时使用索引这个问题，数据库相关文档和书籍也很少或根本没有清晰的说明，开发人员通常只能猜测如何有效地使用索引。

13.2 反模式：无规划地使用索引

当我们通过猜测来选择索引时，不可避免地会犯一些错误。对何时使用索引的误解可能会导致如下三种类型的错误：

- 不使用索引或索引不足；
- 使用了太多的索引或者使用了一些无效索引；
- 执行一些让索引无能为力的查询。

13.2.1 无索引

我们通常都知道，数据库在保持索引同步的时候会有额外的开销。我们每次使用 `INSERT`、`UPDATE`，或者 `DELETE` 时，数据库就不得不更新索引的数据结构来使得所记录的表数据是一致的，然后我们使用这个索引进行查询时就能准确地找到所需要的记录。

我们已经习惯性地将额外开销认为是浪费。因此当我们知道数据库在保持索引同步的时候会造成额外的开销，就想要消除它。一些开发人员于是得出结论，终极的解决方案就是不使用索引。这个建议很常见，但它忽略了一个事实，那就是索引能够通过带给你更多的好处来抵销它的额外开销。

索引不是标准

你知道 ANSI SQL 的标准中没有任何和索引相关的描述吗？数据存储方面的实现和优化在 SQL 语言中并没有明确的说明，因此每个数据库产品在实现索引的方式上都有很高的自由度。

大多数数据库产品都使用相似的 CREATE INDEX 语句，但每个产品都有一定的灵活度来修改并加入一些它们自己的技术。索引没有标准的兼容模式。同样地，也没有索引维护、自动化查询优化、查询计划报表的标准，或者类似于 EXPLAIN 这样的指令。

要尽可能地了解索引，你只能仔细阅读你所使用的数据库产品的文档。具体的语法和特性对于不同的产品来说有很大的区别，但基本的逻辑和理论都是通用的。

不是所有的额外开销都是浪费。你的公司所雇佣的管理层、法律顾问、会计，那些与办公设施有关的开销，甚至那些对公司收入没有直接帮助的开销都是浪费吗？不，因为这些人和物都从不同的方面为公司做了重要的贡献。

在传统的软件中，每一次更新都会执行上百次表查询操作。每次当你执行一个使用索引的查询时，你就抵消了一部分由于维护索引而造成的额外开销。

索引也能帮助 UPDATE 或者 DELETE 语句快速地找到对应的记录。比如，bug_id 这个主键上的索引能有效提升下面这条语句的效率：

Index-Shotgun/anti/update.sql

```
UPDATE Bugs SET status = 'FIXED' WHERE bug_id = 1234;
```

一条在非索引列上执行的语句会导致数据库执行全表遍历来查找匹配记录。

Index-Shotgun/anti/update-unindexed.sql

```
UPDATE Bugs SET status = 'OBSOLETE' WHERE date_reported < '2000-01-01';
```

13.2.2 索引过多

你只能在使用索引进行查询时才能受益。对于那些你不使用的索引，你无法获得任何好处。这里有一些例子：

Index-Shotgun/anti/create-table.sql

```
CREATE TABLE Bugs (
  bug_id        SERIAL PRIMARY KEY,
  date_reported DATE NOT NULL,
  summary       VARCHAR(80) NOT NULL,
  status        VARCHAR(10) NOT NULL,
```

```
    hours         NUMERIC(9,2),
❶   INDEX (bug_id),
❷   INDEX (summary),
❸   INDEX (hours),
❹   INDEX (bug_id, date_reported, status)
  );
```

在之前的例子里，有这么几个无用的索引：

❶ bug_id：大多数数据库都会自动地为主键建立索引，因此额外再定义一个索引就是一个冗余操作。这个额外定义的索引并无任何好处，它只会成为额外的开销。关于何时自动创建索引，每个数据库产品都有自己的规则。你需要仔细地阅读你所使用的数据库的说明文档。

❷ summary：对于长字符串，比如 VARCHAR(80)这种类型的索引要比更为紧凑数据类型的索引大很多。同样地，你也不太可能对 summary 列进行全匹配查找。

❸ hours：这是另一个你不太可能按照特定值进行搜索的列。

❹ bug_id，date_reported，status：使用组合索引是一个很好的选择，但大多数人创建的组合索引通常都是冗余索引或者很少使用。同样地，组合索引中的列的顺序也很重要：你需要在查询条件、联合条件或者排序规则上使用定义索引时的从左到右的顺序。

对冲风险

比尔·科斯比说了一个他在拉斯维加斯的故事。他在赌场输了钱之后觉得很有挫败感，于是决定在走之前至少要赢一点东西。因此他买了 200 美元的 25 分硬币筹码，然后走到了轮盘赌桌前，在每个方格内，不管红的或是黑的，都放了些筹码，把整张赌桌都覆盖了。然后庄家转动了那个球……那个球掉在了地上。

有些人为每一列——以及每个组合列——都创建索引，因为他们不知道哪个索引对查询有帮助。如果你对数据库中的每张表每个列都做索引，就造成了很多无法确定收益的额外开销。

13.2.3 索引也无能为力

接下来常犯的一个错误就是进行一个无法使用索引的查询。开发人员创建了越来越多的索引，尝试着去发现一个神奇的组合方式或者索引选项来加速他们的查询。

我们可以想象数据库的索引使用了类似于电话号码簿的结构。如果我让你去电话号码簿里查一下有哪些人姓 Charles，这会是一个很简单的任务。所有同姓的人都列在了一起，因为这就是电话号码簿排序的方式。

然而，如果我让你去查一下谁的名字叫做 Charles，电话簿的名字排序方式就帮不上忙了。任何人都可以叫 Charles，而不管他姓什么，因此你必须一行行地在号码簿中查找。

电话号码簿是按照先姓后名的方式对联系人进行排序的，就像一个按照 last_name、first_name 顺序创建的联合索引一样。这种索引不能帮你按照名来进行快速查找。

Index-Shotgun/anti/create-index.sql

```
CREATE INDEX TelephoneBook ON Accounts(last_name, first_name);
```

下面有一些无法使用索引进行查询的例子。

- SELECT * FROM Accounts ORDER BY first_name, last_name;
  这个查询就是之前电话号码簿的情况。如果你创建了一种先 last_name 再 first_name 顺序的联合索引（就如同电话号码簿），它是不会帮你先按照 first_name 进行排序。

- SELECT * FROM Bugs WHERE MONTH(date_reported) = 4;
  即使你为 date_reported 列创建了一个索引，这个索引的排序规则也无法帮你按照月份查询。这个索引是按照完整的日期进行排序的，从年份开始。但每一年都有第四个月，因此，月份为 4 的数据分散在整张表中。
  一些数据库支持表达式索引，或者针对衍生列进行和普通列一样的索引。但你必须在使用前明确地定义索引的行为，并且这些索引只能在你定义的表达式查询上节省时间。

- SELECT * FROM Bugs WHERE last_name = '*Charles*' OR first_name = '*Charles*';
  我们又回到之前那个问题上来了，包括指定名字的行分散在整张表中，无法和我们定义的索引顺序匹配。上面的这个查询和下面的这个查询的结果是一样的：

  ```
  SELECT * FROM Bugs WHERE last_name = 'Charles'
  UNION
  SELECT * FROM Bugs WHERE first_name = 'Charles';
  ```

  该例中的索引能帮助我们快速地按姓查找，但对于按名查找则无能为力。

- SELECT * FROM Bugs WHERE description LIKE '*%crash%*';
  由于这个查询断言的匹配子串可能出现在该字段的任何部分，因此即使经过排序的索引结构也帮不上任何忙。

13.3 如何识别反模式

如下几点是使用了“乱用索引”反模式的特征。

- “这是我的查询语句，我要怎样才能让它更快？”
  这可能是最常见的一个关于 SQL 的问题了，但它缺少了表结构、索引、数据集和性能及优化尺度的细节。没有这些上下文，任何的解答都是臆测。
- “我在每个字段上都定义了索引，为什么它没有变得更快？”
  这是“乱用索引”反模式的经典案例。你尝试了所有的索引，但你是在摸黑打鸟啊！

- “我听说索引会使数据库变慢，所以我不会使用它。”
  就像很多开发人员那样，你在找一个提升性能的万全之策。根本没有这样的好事啊！

13.4 合理使用反模式

如果你需要设计一个普通用途的数据库，不了解哪些查询是需要重点优化的，你就不能确定哪些索引是最好的。你可能需要做一些有根据的猜测。你有可能会漏掉一些能有所帮助的索引，也有可能会创建了一些没用的索引。但你必须尽量去尝试。

低分离率索引

分离率是衡量数据库索引的一个指标。它是一张表中，所有不重复的值的数量和总记录条数之比：

```
SELECT COUNT(DISTINCT status) /
    COUNT(status) AS selectivity FROM Bugs;
```

分离率的值越低，索引的效率就越低。为什么？我们可以想象下面这样的情况。

这本书有一个关于不同数据类型的索引：索引中的每一条记录都给出这个单词出现的页的列表。如果一个单词在这本书中频繁地出现，就可能会有很多页码。要找到所需要找到的部分，你就不得不翻到这个列表中的每一页去查看。

索引本身并不会觉得存储那些出现频率很高的单词有什么负担。但如果你需要频繁地在索引和页面间来回查找所需要的内容，你可能就跟从头到尾读了这本书没什么区别。

数据库的索引机制也是类似的，如果一个给定的值出现在这张表的很多条记录中，查询索引比简单地扫描一遍整张表更麻烦。事实上，使用这个索引的开销可能比不使用它还大。

你需要时刻关注你的数据库中索引的分离率，并且抛弃那些低效的索引。

13.5 解决方案：MENTOR 你的索引

“乱用索引”这个反模式是关于随意创建或抛弃索引的，因此，让我们来分析一下一个数据库，并且找一些好的理由来使用或者丢弃索引。

你可以使用好记的 MENTOR 方法来分析数据库索引的使用：测量（Measure），解释（Explain），挑选（Nominate），测试（Test），优化（Optimize）和重建（Rebuild）。

数据库不一定是瓶颈

软件开发人员通常的经验是数据库总是程序中最慢的部分，并且是性能问题的根源。然而，事实并非如此。

举例来说，在我曾经参与过的一个项目中，我的经理让我找出为什么程序跑得这么慢，并且他坚持认为是数据库的错。然而在我用了一个性能测评工具去检测程序代码之后，我发现它用了 80%的时间来解析 HTML，然后找出表单字段并往里面填入对应的值。这个性能问题和数据库完全无关。

在对性能问题下结论之前，先用一些分析工具来测试一下。否则你可能就在做一些过度优化。

13.5.1 测量

你不能在没有信息的情况下做出决定。大多数数据库都提供了一些方法来记录执行 SQL 查询的时耗，因此可以以此来定位最耗时的查询。

- Oracle 和微软的 SQL Server 都有 SQL 跟踪功能和工具来生成并分析相应报表。微软称之为 SQL Server Profiler，Oracle 称之为 TKProf。
- MySQL 和 PostgreSQL 会记录耗时超过一个特定值的查询请求。MySQL 称之为“慢查询”日志，其配置文件中的 `long_query_time` 项默认设定为 10 秒。PostgreSQL 有一个类似的配置项叫做 `log_min_duration_statement`。
- PostgreSQL 还有一个配套的工具叫做 pgFouine，它能帮助你对查询日志进行分析,并且定位出那些需要格外注意的查询请求（http://pgfouine.projects.postgresql.org/）。

一旦你知道程序中哪些查询耗时最多，就知道该专注于哪方面的优化才能获得最大的效果。你甚至可能发现除了某一个特定的查询比较慢之外，其他的都很快，那这个查询就是性能的瓶颈。这个查询就是你该优化的对象。

如果一个查询很少被调用，那即使它是单次调用耗时最多的一个，也不见得是最耗时间的查询。其他更简单一点的查询可能被调用得很频繁，比你所预期的还要频繁，因此它们所花费的总时间就会更多。专注于这些能够让你事半功倍的查询优化。

记住在做查询性能测试的时候要禁止所有的查询结果缓存。这些缓存被设计用来绕过查询过程和索引使用，因此，如果不禁止这些缓存，你是得不到准确信息的。

在部署程序之后，你可通过进行 profile 分析来得到更准确的信息。要在实际用户的使用中收集综合数据来查看到底在哪些地方所花的时间是最长的。你应该时刻监控着这些 profile 数据，以防止不小心造成了一个新的瓶颈。

记住在分析完了之后要关闭 profiler 或者降低 profiler 运行的频率，因为这些工具也会造成额外的开销。

13.5.2 解释

已经找到耗时最多的查询请求了，接下来要做的事情就是找出它之所以会这么慢的原因。每个数据库都使用一种优化工具为每次查询选择合适的索引。你可以让数据库生成一份它所做分析的报表，我们称之为*查询执行计划*（QEP）。

每种数据库的请求 QEP 的语法都不尽相同。

数据库	QEP报表方案
IBM DB2	EXPLAIN, db2expln命令, 或 Visual Explain
Microsoft SQL Server	SET SHOWPLAN_XML, 或 Display Execution Plan
MySQL	EXPLAIN
Oracle	EXPLAIN PLAN
PostgreSQL	EXPLAIN
SQLite	EXPLAIN

QEP 的报表中包含什么或者报表的形式是什么，没有统一标准。通常来说，QEP 会告诉你在一个查询中需要用到哪些表，优化工具是怎么选择索引的，以及按照什么顺序访问这些表。报表可能也会包含一些统计信息，比如每一阶段查询产生多少行记录等。

让我们来看一个简单的 SQL 查询并请求 QEP 报表：

Index-Shotgun/soln/explain.sql

```
EXPLAIN SELECT Bugs.*
FROM Bugs
JOIN (BugsProducts JOIN Products USING (product_id))
  USING (bug_id)
WHERE summary LIKE '%crash%'
  AND product_name = 'Open RoundFile'
ORDER BY date_reported DESC;
```

图 13-1 为 MySQL 的 QEP 报告，`key` 这一列表明这个查询仅使用了 `BugsProducts` 表的主键索引。同时，最后一列的额外信息表明这个查询会在一张临时表中对数据进行排序，没有任何索引优化。

`LIKE` 表达式强制在 `Bugs` 表中进行全表遍历，在 `Products.product_name` 列上没有索引。我们可以通过在 `product_name` 上创建一个新的索引以及改用全文搜索的方案来优化这个查询[①]。

① 参考第 17 章内容。

table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
Bugs	ALL	PRIMARY,bug_id	NULL	NULL	NULL	4650	100	Using where; Using temporary; Using filesort
BugsProducts	ref	PRIMARY,product_id	PRIMARY	8	Bugs.bug_id	1	100	Using index
Products	ALL	PRIMARY,product_id	NULL	NULL	NULL	3	100	Using where; Using join buffer

图 13-1　MySQL 查询执行计划

QEP的信息是由数据库开发商决定的。在本例中，你应该仔细阅读MySQL手册中“Optimizing Queries with EXPLAIN”一章中的说明来理解对应的报表的含义[①]。

13.5.3　挑选

现在已经有了查询优化工具的 QEP 报表，你应该仔细地查找那些没有使用索引的查询操作。

有些数据库会用一些工具来做这件事，它们会收集查询统计信息并且提出一系列的修改建议，包括创建那些被你漏掉但能起到较好效果的索引。比如：

- IBM DB2 Design Advisor；
- Microsoft SQL Server Database Engine Tuning Advisor；
- MySQL Enterprise Query Analyzer；
- Oracle Automatic SQL Tuning Advisor。

即使没有自动化工具，你也可以学习如何辨识一个索引是否有利于提高搜索效率。你需要仔细阅读你所使用数据库的手册来更好地理解 QEP 报告。

索引覆盖

如果一个索引包含了我们所需要的所有列，那就不需要再从表中获取数据了。

想象一下，如果电话簿的条目中只包含一个页码，在你找到一个名字之后，你不得不翻到对应的页上才能得到所需要的号码。如果将这个过程整合为单步操作会更合情合理。由于电话簿是排序的，所以查找一个名字是很快的，然后在一个条目中可以再包含一个字段存储电话号码，甚至存储地址。

这就是索引覆盖的作用。你可以定义让一个索引包含额外的列，即使这些列对于这个索引来说并不是必须包含的。

① http://dev.mysql.com/doc/refman/5.1/en/using-explain.html。

```
CREATE INDEX BugCovering ON Bugs
  (status, bug_id, date_reported, reported_by, summary);
```

如果你查询的列都包含在这个索引的数据结构中，数据库就可以通过只查询索引生成结果。

```
SELECT status, bug_id, date_reported, summary
FROM Bugs WHERE status = 'OPEN';
```

数据库并不需要去查询对应的表。虽然你不能在任何地方都使用索引覆盖，但只要使用它，对性能来说就是一个极大的提升。

13.5.4 测试

这一步非常重要：在创建完索引之后，需要重新跟踪那些查询。需要确认你的改动确实提升了性能，然后就能确定工作完成了。

你可以使用这一步骤来给老板留下好印象，并证明你所作的优化是有效的。你肯定不希望周报上写道："我尝试了每个我想到的办法来解决程序的性能问题，然后我们只能等等看反馈……"相反，你现在有机会这么写周报："我发现可以在一个高活跃度的表上创建一个新的索引，并且我将核心查询性能提高了 38%。"

13.5.5 优化

索引是小型的、频繁使用的数据结构，因而很适合将它们常驻在内存中。内存操作的性能是磁盘 I/O 操作的好几倍。

数据库服务允许你配置缓存所需要的系统内存大小。大多数数据库的默认配置都很小，从而能保证数据库在大部分操作系统上都正常工作。通常情况下我们需要调高这个缓存大小的设置。

需要使用多少内存做为索引缓存？这个问题没有标准答案，因为它取决于你的数据库的规模和服务器的内存大小。

使用索引预载入的方法可能要比通过数据库活动本身将最频繁使用的数据与索引放入缓存更有效一点。比如，在 MySQL 中，使用 `LOAD INDEX INTO CACHE` 语句。

13.5.6 重建

索引在平衡的时候其效率最高，当你更新或者删除记录时，索引就逐渐变得不平衡，就如同文件系统随着时间的推移会产生很多磁盘碎片一样。在实际运行中，你可能看不出一个平衡索引和一个有些不平衡的索引的区别。但我们想要最大限度地使用索引，因此要定期对索引进行维护。

就像索引的很多其他特性一样，每个不同的数据库都使用自己特有的术语、语法和功能。

数据库	索引维护命令
IBM DB2	REBUILD INDEX
Microsoft SQL Server	ALTER INDEX ... REORGANIZE, ALTER INDEX ... REBUILD, or DBCC DBREINDEX
MySQL	ANALYZE TABLE or OPTIMIZE TABLE
Oracle	ALTER INDEX ... REBUILD
PostgreSQL	VACUUM or ANALYZE
SQLite	VACUUM

多久需要重建一次索引？你可能听到诸如“每周一次”这样空泛的回答，但事实上对于不同的程序来说并没有统一的答案。具体的时间取决于你对指定的表所做的会引起不平衡操作的频率。同时也取决于这张表有多大以及理想状态的索引是否那么的重要。花上几个小时重建一张很大但很少使用的表的索引，只获得了 1%的性能提升，这样做值得吗？所有的判断都需要你来做决定，因为你是最了解这些数据和数据操作需求的人。

很多关于最优化使用索引的知识都是根据不同数据库而论的，因此你需要仔细研究所使用的数据库。你手上所有的资源包括数据库手册、书籍和杂志、博客文章和邮件列表，以及很多自己的经验。最重要的规则是千万别瞎猜索引的使用方法。

了解你的数据，了解你的查询请求，然后 MENTOR 你的索引。

第 14 章

对未知的恐惧

就像我们所知道的那样，有一些众所周知的事情，我们知道自己已经了解了。我们也知道，有一些未知的事情，我们知道自己还不了解。但还有些没人知道的事情，我们并不知我们还一无所知。

► *唐纳德·拉姆斯菲尔德*

在我们的范例 Bug 数据库中，`Accounts` 表有 `first_name` 和 `last_name` 两列。你可以通过使用字符串链接操作将这两个字段合并成用户的全名。

Fear-Unknown/intro/full-name.sql

```
SELECT first_name || ' ' || last_name AS full_name FROM Accounts;
```

假设老板让你修改一下数据库，加上用户的中间名的缩写。（如果两个用户的名和姓是一样的，通过中间名的缩写能够减少误解。）这是个很简单的活。你也可以手动加了一些用户的中间名缩写。

Fear-Unknown/intro/middle-name.sql

```
ALTER TABLE Accounts ADD COLUMN middle_initial CHAR(2);

UPDATE Accounts SET middle_initial = 'J.' WHERE account_id = 123;
UPDATE Accounts SET middle_initial = 'C.' WHERE account_id = 321;

SELECT first_name || ' ' || middle_initial || ' ' || last_name AS full_name
FROM Accounts;
```

突然，程序不显示任何名字了。事实上，看了一下之后，你发现并不都如此。只有那些填了中间名缩写的用户的名字能正常显示，其他人的名字都是空的。

其他人的名字怎么了？你能在老板发现并开始恐慌地猜测你是不是丢了数据之前修复这个错误吗？

14.1　目标：辨别悬空值

不可避免地，你的数据库中总会有一些字段是没有值的。不管是插入一个不完整的行，还是有些列可以合法地拥有一些无效值。SQL 支持一个特殊的空值，就是我们所熟知的 NULL。

有很多在 SQL 的表和查询中有效使用空值的途径。

- 你可以在添加一条记录时，使用 NULL 代替那些还不确定的值，比如一个在职员工的离职时间。
- 一个给定的列如果没有合适的值，可以在对应的行中使用 NULL。比如对于一辆完全靠电力驱动的车，它的燃油消耗比就毫无意义。
- 当传入参数无效时，一个函数的返回值也可以是 NULL，比如 DAY('2009-12-32')。
- 在外联结查询中，NULL 被用来当做未匹配的列的占位符。

本章的目的就是弄清楚如何编写那些包含 NULL 的查询。

14.2　反模式：将 NULL 作为普通的值，反之亦然

很多开发人员都对 SQL 中 NULL 的行为感到茫然无措。SQL 将 NULL 当做一个特殊的值，不同于 0、false 或者空字符串，这一点和大多数的编程语言都不同。大多数的数据库都遵循这种 SQL 标准，然而在 Oracle 和 Sybase 中，NULL 的意义是长度为 0 的空字符串。NULL 这个值也有一些特殊的行为。

14.2.1　在表达式中使用NULL

让人奇怪的是，在一些值为 NULL 的列上进行计算所得到的结果。比如说，很多开发人员都觉得如下查询请求的结果中 10 应当作为 Bugs 的默认值返回，但是当 hours 列中的值为 NULL 是，返回的结果还是 NULL，并非 10。

Fear-Unknown/anti/expression.sql

```
SELECT hours + 10 FROM Bugs;
```

NULL 和 0 是不同的。比未知数大 10 的数还是未知数。

NULL 和空字符串也是不一样的。将一个字符串和标准 SQL 中的 NULL 联合起来的结果还是 NULL（忽略 Oracle 和 Sybase 中的行为）。

NULL 和 FALSE 也是不同的。AND、OR 和 NOT 这三个布尔操作如果涉及 NULL，其结果也让很多人感到困惑。

14.2.2 搜索允许为空的列

如下查询仅返回 assigned_to 为 123 的行，不包含别的值或者 NULL：

Fear-Unknown/anti/search.sql

```
SELECT * FROM Bugs WHERE assigned_to = 123;
```

你可能会觉得如下查询会返回上面那个查询的补集，也就是所有之前查询没有返回的行：

Fear-Unknown/anti/search-not.sql

```
SELECT * FROM Bugs WHERE NOT (assigned_to = 123);
```

然而，这两个查询都不会返回 assigned_to 是 NULL 的记录。任何和 NULL 的比较都返回“未知”，既不是 TRUE 也不是 FALSE。即使 NULL 的相反值也是 NULL。

下面这个错误是在查询 NULL 或者非 NULL 值时常犯的：

Fear-Unknown/anti/equals-null.sql

```
SELECT * FROM Bugs WHERE assigned_to = NULL;

SELECT * FROM Bugs WHERE assigned_to <> NULL;
```

WHERE 子句的筛选逻辑是当其条件的返回值为 TRUE 时选择该记录，但和 NULL 比较永远得不到 TRUE，相应地，返回的是“未知”。无论比较的逻辑是相等还是不等，返回的结果还是“未知”，当然“未知”不等于 TRUE。之前的所有查询请求都没办法获得 assigned_to 为 NULL 的记录。

14.2.3 在查询参数中使用NULL

在参数化的 SQL 查询表达式中使用 NULL 进行查询时，碰到使用 NULL 作为普通值的列也非常地不方便。

Fear-Unknown/anti/parameter.sql

```
SELECT * FROM Bugs WHERE assigned_to = ?;
```

上述查询在传入一个普通的整形时会返回你所期望的值，但你不能使用 NULL 作为参数传入。

14.2.4 避免上述问题

处理 NULL 会使查询变得更加复杂，很多软件开发人员就会选择在数据库中禁止 NULL。取而代之的是，他们选择一个普通的值来标记“未知”或者“无效”。

“我们痛恨NULL！”

杰克，一个软件开发人员，描述了他的客户端开发人员要求他屏蔽数据库中

所有的 NULL。他们的解释就是简单的“我们痛恨 NULL”。NULL 的存在会导致他们的程序代码出错。杰克询问我该用哪个值来代替 NULL 标记悬空值。

我告诉杰克，NULL 的作用正是用来表示悬空值或者无效值。无论他选择别的什么值来标记一个悬空值，都要修改程序代码来对这个值进行特殊处理。

杰克的客户端开发人员对待 NULL 的态度是完全错误的。类似地，我可以坦率地说，我也不喜欢写代码来处理将零作为除数的问题，但这并不是不使用零的合理理由。

这样做的实际危害在哪里呢？来看如下的例子，我们将 assigned_to 这一列声明为 NOT NULL：

Fear-Unknown/anti/special-create-table.sql

```
CREATE TABLE Bugs (
  bug_id            SERIAL PRIMARY KEY,
  -- other columns
  assigned_to       BIGINT UNSIGNED NOT NULL,
  hours             NUMERIC(9,2) NOT NULL,
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);
```

假设我们使用-1 表示一个未知的值。

Fear-Unknown/anti/special-insert.sql

```
INSERT INTO Bugs (assigned_to, hours) VALUES (-1, -1);
```

hours 这一列是数字，因此你定义某一个特殊的数字表示“未定义”。这个数字在这列中必须不具有任何含义，所以你选择了一个负数。但是-1 这个值在执行 SUM 或者 AVG 的时候会导致计算结果错误。因而你必须在计算时使用另一个条件判断来去除这些列，这些额外的工作都是因为你要避免使用 NULL 所带来的。

Fear-Unknown/anti/special-select.sql

```
SELECT AVG( hours ) AS average_hours_per_bug FROM Bugs
WHERE hours <> -1;
```

在另一列中，-1 可能是一个有意义的值，因此你不得不根据每一列的定义和取值范围逐个选择一个不同的值。你还需要记住或者将这些特殊值写成文档。这都给项目增加了过多的复杂度和不必要的工作。

再来看 assigned_to 列。这是一个指向 Accounts 表的外键。当一个 Bug 被提交进系统但还没有指派给任何人去处理，应该用哪个非 NULL 的值来表示这个逻辑呢？任何一个非 NULL 的值都必须指向 Accounts 表中的一条记录，因此你需要在 Accounts 表中插入一条占位符似的记录，表示“没有人”或者“未指派”。为了让你能将一个 Bug 指派给一个人而创建这样一个没有意义的账号看上去非常地可笑。

当你将一列声明为 NOT NULL 时，也就是说，这列中的每一个值都必须存在且是有意义的。

比如说，Bugs.reported_by 这一列必须有一个值，因为每个 Bug 都是由某个人报告的。但一个 Bug 可能暂时还没有指派给任何人处理。悬空的值应该用 NULL 来表示。

14.3 如何识别反模式

如果你发现自己或者团队中有人描述了如下事件，可能就是没有正确地处理 NULL。

- “我要怎么将 assigned_to（或别的列）中没有值的列取出来？”
  你不能对 NULL 使用等号。我们在本章的后半部分将会看到如何使用 IS NULL 操作符。
- “我能在数据库中看到用户的名字，但是在程序显示的时候，全名就是空的。”
  问题可能是由于你将字符串和 NULL 进行了拼接的操作，返回的结果还是 NULL。
- “这份报告中这个项目的总工作时间仅包括了很小一部分我们设定了优先级的 Bug!”
  你的统计时间的合计查询可能包含了一个 WHERE 子句，其中的表达式会因为 priority 为 null 而不会返回 TRUE。当你使用“不等于”操作的时候要格外注意异常值。比如，对于 priority 的值为 NULL 的记录，priority <> 1 这个表达式就不会返回真。
- “现在的情况是，我们不能再使用之前在 Bugs 表中用来表示未知的那个字符串了，因此我们要开个会讨论该用哪个新的特殊值，再评估一下开发时间以及数据转换的时间。”
  这是使用一个特殊标记值的常见后果，这个特殊值很有可能会变得合法。最终你可能会发现需要使用这个值的字面意义而不是标记意义。

NULL 也是可关联的吗

关于 SQL 中的 NULL 也有一些争论。创立关系理论的计算机科学家 E. F. Codd 认为，使用 NULL 来标记悬空值是有必要的。然而，C. J. Date 展示了标准 SQL 中定义的 NULL 的行为在某些特殊情况下和关系逻辑相违背。

事实是大多数编程语言都没有完美地实现计算机科学的那些理论。无论如何 SQL 都支持 NULL。我们已经发现了一些危害，但你可以学着如何找出这些危害从而更有效地使用 NULL。

要辨识出对 NULL 的错误使用是比较困难的事情。在测试阶段，一些错误可能不会发生，尤其是如果你漏掉了一些边界条件的测试而仅仅构造了一些简单的测试数据。然而，当程序用于实际生产时，所产生的数据可能是你不曾预料到的。如果一个 NULL 存在于数据中，产生错误就是一个时间问题了。

14.4 合理使用反模式

使用 NULL 并不是反模式，反模式是将 NULL 作为一个普通值处理或者使用一个普通的值来取代 NULL 的作用。

有一种情况可以将 NULL 视为普通值，那就是导入或者导出数据的时候。在一个以逗号分割的文本文件中，所有的值都必须是可读的文本。比如，MySQL 的 mysqlimport 工具在从文本文件中导入数据时，使用\N 代表 NULL。

类似地，用户不能直接输入一个 NULL。程序的输入端可能会使用一些特殊处理来引导用户输入 NULL。比如，微软.NET 2.0 及以上版本，为 Web 界面提供了一个叫做 ConvertEmptyStringToNull 的方法。参数和绑定字段会自动地将空字符串转换成 NULL。

最后，如果需要支持多个不同的悬空值时，NULL 也不起作用。比如说，想要将一个 Bug 分为“从未被指派”和“被指派给一个离开项目组的人”，在这种情况下你不得为每种状态使用不同的值。

14.5　解决方案：将 NULL 视为特殊值

大多数使用 NULL 的问题都是源自于对 SQL 的三值逻辑的误解。对于习惯了传统的 true/false 逻辑程序员来说，这可是一个不小的挑战。不过只要稍微研究一下，你就能正确地在 SQL 中处理 NULL。

14.5.1　在标量表达式中使用NULL

假设斯坦 30 岁，而奥利弗的年龄未知。如果我问你到底是斯坦大还是奥利弗大，你只能回答：“我不知道。”如果我问你斯坦是不是和奥利弗一样大，你还是只能回答：“不知道。”如果我问你他们两个加起来有多大，你的答案依旧如此。

假设查理的年龄也是未知。如果我问你奥利弗和查理是不是一样大，你的答案还是“不知道”。这就是为什么 NULL = NULL 的结果是 NULL。

下表列举了一些程序员期望得到某种结果，但事实却不如人意的情况。

表达式	期望值	实际值	原　因
NULL = 0	TRUE	NULL	NULL不是0
NULL = 12345	FALSE	NULL	如果未指定值和所给值相等则未知
NULL <> 12345	TRUE	NULL	不相等则未知
NULL + 12345	12345	NULL	NULL不是0
NULL || 'string'	'string'	NULL	NULL不是空字符串
NULL = NULL	TRUE	NULL	未指定值和另一个值相等则未知
NULL <> NULL	FALSE	NULL	如不同则未知

当然，这些例子并不仅仅表示直接使用 NULL 这个关键字的情况，它们同样也适用于那些返回值为 NULL 的表达式。

14.5.2 在布尔表达式中使用NULL

理解 NULL 在布尔表达式中行为的关键点在于，要明白 NULL 既不是 TRUE 也不是 FALSE。

下表列举了一些程序员所期望得到某种结果，但事实却不如人意的情况。

表达式	期望值	实际值	原　因
NULL AND TRUE	FALSE	NULL	NULL不是FALSE
NULL AND FALSE	FALSE	FALSE	任何值AND FALSE是伪值
NULL OR FALSE	FALSE	NULL	NULL不是FALSE
NULL OR TRUE	TRUE	TRUE	任何值OR TRUE是真值
NOT (NULL)	TRUE	NULL	NULL不是FALSE

一个 NULL 值当然不是 TRUE，同样也不是 FALSE。如果是 FALSE，对一个 NULL 使用 NOT 操作符，则应该返回 TRUE，但事实是 NOT(NULL)依旧返回一个 NULL。这样的行为让那些想要在布尔表达式中使用 NULL 的人感到很困惑。

14.5.3 检索NULL值

由于等于或者不等于操作在对 NULL 进行比较时都不会返回 TRUE，你就需要使用别的操作来检索 NULL。老的 SQL 标准定义了一个 IS NULL 的断言，在一个给定的操作值为 NULL 时，它将返回 TRUE。与之对应的，IS NOT NULL 在操作值是 NULL 时，返回 FALSE。

Fear-Unknown/soln/search.sql

```
SELECT * FROM Bugs WHERE assigned_to IS NULL;

SELECT * FROM Bugs WHERE assigned_to IS NOT NULL;
```

在 SQL-99 标准中，额外定义了一个比较断言 IS DISTINCT FROM。这个断言的行为模式类似于原来的<>操作符。不同的是，在处理操作数为 NULL 时它依旧会返回 TRUE 或 FALSE。

错误的方法正确的结果

在如下的情况中，一个允许为空的列的行为基于巧合而得到了正确的结果。

```
SELECT * FROM Bugs WHERE assigned_to <> 'NULL';
```

这里允许为空的 assigned_to 列正用来和'NULL'这个字符串进行比较（注意引号），而不是和 NULL 这个关键字比较。

当 assigned_to 为 NULL 时，和字符串'NULL'的比较结果不为 TRUE。这一条记录就从查询结果中排除了，而这正是程序员所期望的。

其余的情况是，一个整型的列和字符串'NULL'进行比较。像'NULL'这样的字符串在大多数数据库中都被视为 0，而 assigned_to 的大多数值都大于 0。这样的值就不等于一个字符串，因此在结果集中就有了这条记录。

因此，由于犯了另一个常见的错误——在 NULL 关键字上使用了引号——一些程序员可能无意间就得到了他们想要的结果。不幸的是，这样的巧合并不是总发生，比如 WHERE assigned_to = 'NULL'。

这个断言能让你在进行比较操作之前不用再编写冗长的表达式判断 IS NULL。如下两个查询是等价的：

Fear-Unknown/soln/is-distinct-from.sql

```
SELECT * FROM Bugs WHERE assigned_to IS NULL OR assigned_to <> 1;

SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM 1;
```

你可以和查询参数一起使用这个断言，即使传入值就是一个 NULL：

Fear-Unknown/soln/is-distinct-from-parameter.sql

```
SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM ?;
```

每个数据库对 IS DISTINCT FROM 的支持是不同的。PostgreSQL、IBM DB2 和 Firebird 直接支持它，Oracle 和 Microsoft SQL Server 暂时还不支持。MySQL 提供了一个专有的表达式<=>，它的工作逻辑和 IS NOT DISTINCT FROM 一致。

14.5.4 声明NOT NULL的列

如果 NULL 会破坏程序逻辑或者 NULL 本身就是毫无意义的，那我推荐你在定义对应的列时加上 NOT NULL 约束。让数据库来帮你确保约束的实行比自己写代码可靠得多。

比如说，在 Bugs 表中任意记录的 date_reported、reported_by 和 status 这几列的值都应该是非 NULL 的。类似地，在 Comments 这张子表中也必须有一个非 NULL 的 bug_id 列，指向一

个切实存在的 Bug。你应该在声明这些列的时候都加上 NOT NULL 选项。

有些人推荐你为每一列都定义一个 DEFAULT 值，这样一来当在执行插入操作时即使省略了某一列，也能获得一个非 NULL 的值。这样的建议也并不是通用的。比如说，Bugs.reported_by 应该总是非 NULL 的。如果要定义默认值，应该用哪个值？对于一个在逻辑上没有默认值的列来说，声明一个 NOT NULL 约束是很合理、很常见的。

14.5.5　动态默认值

在一些查询请求中，你需要强制让某一列或者某个表达式返回非 NULL 的值，从而让查询逻辑变得更简单，但又不想将这个值存下来。你所需要的仅仅是在特定的请求时对一个给定的列临时设置一个默认值的方法。为此可以使用 COALESCE()函数。这个函数接受一系列的值作为入参，并且返回第一个非 NULL 的参数。

在本章最开始所描述的拼接用户名字的案例中，可以使用 COALESCE()函数来写一个表达式，使用一个空格代替中名缩写，这样即使中名缩写是 NULL，返回的结果也始终是一个非 NULL 的中间名缩写，从而最终的结果不会变成 NULL。

Fear-Unknown/soln/coalesce.sql

```
SELECT first_name || COALESCE(' ' || middle_initial || ' ', ' ') || last_name
  AS full_name
FROM Accounts;
```

COALESCE()是一个 SQL 标准函数。一些数据库使用了别的函数来实现这个功能，诸如 NVL()或 ISNULL()。

使用 NULL 来表示任意类型的悬空值。

第 15 章

模棱两可的分组

智商在于区分可行与否，理性在于区分明智与否。有时候可行却并不明智。

▶ 马克思・伯恩

假设你的老板想通过 Bug 数据库来了解哪些项目还处于活跃状态，哪些项目已经停止了。他让你生成的一个每个项目最后一个 Bug 报告提交日期的报表。你写了个查询过程，在 MySQL 中查询根据 `product_id` 分组的 Bug 的 `date_reported` 最大的值。生成的报表类似于下面这样。

product_name	latest	bug_id
Open RoundFile	2010-06-01	1234
Visual TurboBuilder	2010-02-16	3456
ReConsider	2010-01-01	5678

你的老板是一个注重细节的人，他花了点时间研究这份报告。他注意到，在 Open RoundFile 这个项目下所列出来的最新的 `bug_id` 其实并不是最新的。比较一下全量数据就能看出差异。

product_name	date_reported	bug_id	
Open RoundFile	2009-12-19	1234	这个bug_id……
Open RoundFile	*2010-06-01*	2248	与这个日期不匹配
Visual TurboBuilder	2010-02-16	3456	
Visual TurboBuilder	2010-02-10	4077	
Visual TurboBuilder	2010-02-16	5150	
ReConsider	2010-01-01	5678	
ReConsider	2009-11-09	8063	

你要怎么解释这个问题？为什么它会只影响了一个产品但对于其他的产品来说又是正常的？你要怎么才能得到正确的报告？

15.1 目标：获取每组的最大值

大多数程序员在学习 SQL 时都会遇到在查询时需要使用 `GROUP BY` 的情况，比如对分组多行

记录使用一些聚合函数，每组获取一条统计记录等。GROUP BY 的强大之处就在于，它把复杂的报表生成过程简化到只用相对很少的代码。

例如，想要使用单次查询来获取 Bug 数据库中每一个产品最新的 Bug 报告，可以这么写：

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

对于上述查询最基本的扩展就是获取最新 Bug 的 ID：

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

然而，这个查询在不同的数据库中，要么是一个语法错误，要么就得到一个不可靠的结果。使用 SQL 的开发人员对此普遍感到很困惑。

本章的目标就是能执行一个不仅返回每组最大值的查询（或者最小值、平均值），同时也要能返回这个值对应的记录的其他字段。

15.2　反模式：引用非分组列

造成这个反模式的最根本原因很简单，而且它揭示了很多程序员对于 SQL 中分组查询逻辑的普遍误解。

15.2.1　单值规则

属于每个组的行，它们的 GROUP BY 关键字所指定的那些列中的值都是一样的。比如下面这个查询，对于每个不同的 product_id 都会返回一条记录。

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

跟在 SELECT 之后的选择列表中的每一列，对于每个分组来说都必须返回且仅返回一个值。这就称为*单值规则*。在 GROUP BY 子句中出现的列能够保证它们在每一组都只有一个值，无论这一个组匹配多少行。

MAX()表达式也能保证每组都返回单一的值，即返回传入 MAX()的参数中最大的那个值。

然而，数据库服务不能确定其他那些在选择列表中的列，它们都是单值的。它不能保证被分

在同一组中的行的其他列都共有一个值。

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

在这个例子中，一个给定的 product_id 有很多不同的 bug_id，因为 BugsProducts 表将很多 Bug 和同一个产品关联起来了。在一个根据 product_id 进行分组的查询中，数据库没办法在查询结果中表示所有的 bug_id。

由于对于这些“额外”的列没有办法保证它们满足单值规则，数据库就假设它们违反了单值规则。大多数的数据库在你尝试返回一个不在 GROUP BY 的参数中的列或者不是由聚合函数返回的值时，抛出一个错误。

MySQL 和 SQLite 在这方面的行为和其他的数据库不同，我们将在 15.4 节中具体说明。

15.2.2 我想要的查询

程序员中常见的误解是认为 SQL 能猜测你需要在报告中显示哪个 bug_id，因为 MAX()是用在另一列而不是 GROUP BY 的那些列上。大多数人假设如果查询得到了最大值，那么查询返回结果中的其他列就会是对应的这个最大值所在的列中的那些值。

不幸的是，SQL 由于下面这几个原因，不会这么做。

- 如果两个 Bug 的 date_reported 值相同并且这个值就是这一组中的最大值，哪个 bug_id 应该放到报告中？
- 如果聚合函数的返回值没有匹配表中的任何一行，又该用哪个 bug_id？当使用 AVG()、COUNT()和 SUM()时，这种事情经常发生。

Groups/anti/sumbyproduct.sql

```
SELECT product_id, SUM(hours) AS total_project_estimate, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

- 如果查询用到了两个不同的聚合函数，比如 MAX()和 MIN()，这可能会定位到一组中两条不同的记录。那这组应该返回哪个 bug_id？

Groups/anti/maxandmin.sql

```
SELECT product_id, MAX(date_reported) AS latest,
  MIN(date_reported) AS earliest, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

这些就是为什么单值规则是如此重要的原因。不是每个不遵照这个规则的查询都会导致模棱两可的结果，但大多数都是这样。如果数据库能够区分有歧义和无歧义查询，并且只有在数据会导致歧义的情况下才返回错误就好了。但这样会导致程序的可靠性降低，这意味着同样的查询可能是合理的，也可能是不合理的，唯一的标准竟然是数据的状态！

15.3 如何识别反模式

对于大多数数据库来说，当输入一个违背了单值规则的查询时，会立刻返回给你一个错误。下面这些就是不同数据库返回的错误信息。

- Firebird 2.1:
  Invalid expression in the **select** list (**not** contained in either an 聚合函数 **or** the GROUP BY clause)
- IBM DB2 9.5:
  An expression starting with "*BUG_ID*" specified in a SELECT clause, HAVING clause, **or** ORDER BY clause is **not** specified in the GROUP BY clause **or** it is in a SELECT clause, HAVING clause, **or** ORDER BY clause with a column function **and** no GROUP BY clause is specified.
- Microsoft SQL Server 2008:
  Column '*Bugs.bug_id*' is invalid in the **select** list because it is not contained in either an 聚合函数 **or** the GROUP BY clause.
- MySQL 5.1，在设置 ONLY_FULL_GROUP SQL 模式之后禁止歧义查询。
  '*bugs.b.bug_id*' isn't in GROUP BY
- Oracle 10.2:
  not a GROUP BY expression
- PostgreSQL 8.3:
  column "*bp.bug_id*" must appear in the GROUP BY clause **or** be used in an 聚合函数

在 SQLite 和 MySQL 中，有歧义的列可能包含不可预测的和不可靠的数据。在 MySQL 中，返回的值是这一组结果中的第一条记录，其排序规则是按照实际的物理存储顺序来定义的。SQLite 的结果正好与 MySQL 相反，它返回最后一条记录。这两者的行为模式都没有文档描述，并且这两个数据库都不保证在后续版本中依旧这么执行。注意这些特征并且合理设计你的查询语句来避免歧义，是你所需要做的。

> **GROUP BY 和 DISTINCT**
>
> SQL 支持另一个查询筛选关键字 DISTINCT，它的作用是对查询结果进行去重操作，这样最终返回的每一行都是唯一的。比如，下面的这个查询返回谁在哪天提交过 Bug，但每个人每天只返回一条记录：
>
> ```
> SELECT DISTINCT date_reported, reported_by FROM Bugs;
> ```
>
> 分组查询如果去掉所有的聚合函数，也能完成同样的事情。通过使用 GROUP BY，也能减少到 GROUP BY 中的列的每一个不同组合都只返回一条查询结果：
>
> ```
> SELECT date_reported, reported_by FROM Bugs
> GROUP BY date_reported, reported_by;
> ```
>
> 这两个查询返回同样的结果，而且其优化和执行过程也应该类似，因此这个例子中唯一的不同大概就是偏好问题了。

15.4　合理使用反模式

正如我们所见的，MySQL 和 SQLite 不能保证那些不满足单值规则的列返回的数据可靠。有些情况下你能利用这种不严谨的规则获得一些好处。

Groups/legit/functional.sql

```
SELECT b.reported_by, a.account_name
FROM Bugs b JOIN Accounts a ON (b.reported_by = a.account_id)
GROUP BY b.reported_by;
```

在之前的查询中，account_name 列从技术上来说违背了单值规则，因为它既没有出现在 GROUP BY 子句中，也没有经过聚合函数的处理。然而，每组中的 account_name 只可能有一个值，这个查询中的分组是依照 Bugs.reported_by 来进行的，而这个列是指向 Accounts 表的一个外键，因此，分组规则满足和 Accounts 表的一对一关系。

换句话说，如果你知道 reported_by 的值，那就可以毫无歧义地断定对应的 account_name，就好像你是根据 Accounts 表的主键进行查询一样。

这种没有歧义的关系叫做函数依赖。最常见的例子就是表的主键和对应的值：account_name 和它的主键 account_id 之间是一个函数依赖。如果你对一张表的主键进行分组查询，那么每一个分组都会定位到表中唯一的一行，因此这一组中的其他列就必然只会有一个值。

Bugs.reported_by 和 Accounts 表中的其他依赖属性有类似的关系，因为它引用了 Accounts 表的主键。当查询是基于 reported_by 的分组时，由于它是一个外键，Accounts 表的属性是函数依赖的，那么查询结果不会产生任何歧义。

然而，大多数的数据库依然会返回一个错误。不仅因为 SQL 标准要求这样的行为，而且在执行中要找出这样的函数依赖的开销也不是很大①。但如果你使用的是 MySQL 或者 SQLite 并且小心谨慎地处理那些函数依赖列上的查询，你就能使用这样的查询并且避免歧义数据。

15.5 解决方案：无歧义地使用列

接下来的几节将介绍几种方法解决这个反模式，教你如何写出不带歧义的查询语句。

15.5.1 只查询功能依赖的列

最直接的解决方案就是将有歧义的列排除出查询。

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

这个查询取出产品最新 Bug 提交的日期，尽管它并不获取最新 Bug 的 `bug_id`。很多时候这就够了，别忽视简单的解决方案。

15.5.2 使用关联子查询

关联子查询会引用外联结查询，并且根据外联结查询中的每一条记录最终返回不同的结果。通过用子查询来搜索同一个产品中 Bug 日期的更新，可以找到每个产品最新的 Bug。只要子查询没有返回，那么外联结查询到的 Bug 就是最新的。

Groups/soln/notexists.sql

```
SELECT bp1.product_id, b1.date_reported AS latest, b1.bug_id
FROM Bugs b1 JOIN BugsProducts bp1 USING (bug_id)
WHERE NOT EXISTS
  (SELECT * FROM Bugs b2 JOIN BugsProducts bp2 USING (bug_id)
   WHERE bp1.product_id = bp2.product_id
     AND b1.date_reported < b2.date_reported);
```

这个查询非常地简单易读。然而，要记住这个查询的性能并不是最好的，因为外联结查询结果中的每一条记录都会执行一遍关联的子查询。

15.5.3 使用衍生表

你可以使用衍生表来执行子查询，先得到一个临时的结果，只包含 `product_id` 和其对应的最新的 Bug 报告日期。然后使用这个临时表和原表进行联结查询，然后就能得到每个产品的最新的 Bug 信息。

---

① 本章的例子都很简单。要确认其他任意的 SQL 查询是不是功能依赖会比这些例子困难得多。

Groups/soln/derived-table.sql

```
SELECT m.product_id, m.latest, b1.bug_id
FROM Bugs b1 JOIN BugsProducts bp1 USING (bug_id)
  JOIN (SELECT bp2.product_id, MAX(b2.date_reported) AS latest
   FROM Bugs b2 JOIN BugsProducts bp2 USING (bug_id)
   GROUP BY bp2.product_id) m
  ON (bp1.product_id = m.product_id AND b1.date_reported = m.latest);
```

product_id	latest	bug_id
1	2010-06-01	2248
2	2010-02-16	3456
2	2010-02-16	5150
3	2010-01-01	5678

注意一点，如果子查询返回的最新日期匹配多个行，那么对应每个产品，你会获得多个行。如果你要确保每个 product_id 仅有一条记录，就可以在外联结查询时使用另一个分组函数：

Groups/soln/derived-table-no-duplicates.sql

```
SELECT m.product_id, m.latest, MAX(b1.bug_id) AS latest_bug_id
FROM Bugs b1 JOIN
  (SELECT product_id, MAX(date_reported) AS latest
   FROM Bugs b2 JOIN BugsProducts USING (bug_id)
   GROUP BY product_id) m
  ON (b1.date_reported = m.latest)
GROUP BY m.product_id, m.latest;
```

product_id	latest	latest_bug_id
1	2010-06-01	2248
2	2010-02-16	5150
3	2010-01-01	5678

衍生表方案是一个相对于关联子查询可扩展性更好的方案。衍生表并不是关联的，因此大多数品牌的数据库都能够一次执行子查询。然而，数据库必须将临时得到的记录存在一张临时表中，因此，这个方案也不是性能最优的方案。

15.5.4 使用JOIN

你可以创建一个联结查询去匹配那些可能不存在的记录。这样的联结查询被称为外联结查询。如果尝试匹配的记录不存在，就会使用 NULL 来替代相应的列。因此，如果查询结果返回了 NULL，我们就知道没有找到相应的记录。

Groups/soln/outer-join.sql

```
SELECT bp1.product_id, b1.date_reported AS latest, b1.bug_id
FROM Bugs b1 JOIN BugsProducts bp1 ON (b1.bug_id = bp1.bug_id)
LEFT OUTER JOIN (Bugs AS b2 JOIN BugsProducts AS bp2 ON (b2.bug_id = bp2.bug_id))
  ON (bp1.product_id = bp2.product_id AND (b1.date_reported < b2.date_reported
```

```
    OR b1.date_reported = b2.date_reported AND b1.bug_id < b2.bug_id))
WHERE b2.bug_id IS NULL;
```

product_id	latest	bug_id
1	2010-06-01	2248
2	2010-02-16	5150
3	2010-01-01	5678

对于大多数人来说，要看明白这个查询需要花点时间，并且需要在草稿纸上做点标记、计算。但是，一旦你弄明白它是怎么回事之后，它会变成一个很重要的工具。

JOIN 解决方案适用于针对大量数据查询并且可伸缩性比较关键时。尽管这个方案比较难以理解和维护，但它总是能比基于子查询的解决方案更好地适应数据量的变化。记住一定要对不同类型的查询的性能进行实际的测量，而不是仅靠猜测来判断哪个更好。

15.5.5　对额外的列使用聚合函数

你可以通过对额外的列使用另一个聚合函数，从而使得它们遵从单值规则。

Groups/soln/extra-aggregate.sql

```
SELECT product_id, MAX(date_reported) AS latest,
  MAX(bug_id) AS latest_bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

只有确定最新的 bug_id 对应的 Bug 的日期也是最新的时候，才能使用这个方案，也就是说，Bug 是按照时间顺序提交的。

15.5.6　连接同组所有值

最后，还有一个聚合函数可以用来处理 bug_id 并避免违背单值规则。MySQL 和 SQLite 提供了一个叫做 GROUP_CONCAT()的函数，它能将这一组中所有的值连在一起作为单一值返回，默认情况下，返回的是一个由逗号分割的字符串。

Groups/soln/group-concat-mysql.sql

```
SELECT product_id, MAX(date_reported) AS latest
  GROUP_CONCAT(bug_id) AS bug_id_list,
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

product_id	latest	bug_id_list
1	2010-06-01	1234,2248
2	2010-02-16	3456,4077,5150
3	2010-01-01	5678,8063

这个查询并不会告诉你哪个 bug_id 对应最新的日期，bug_id_list 包含了这一组中的所有 bug_id。

这个方案的另一个缺点是，它并非 SQL 标准函数。其他的数据库并不支持这个函数。有些数据库支持自定义函数和自定义聚合函数。比如，PostgreSQL 中可以这么处理：

Groups/soln/group-concat-pgsql.sql

```
CREATE AGGREGATE GROUP_ARRAY (
  BASETYPE = ANYELEMENT,
     SFUNC = ARRAY_APPEND,
     STYPE = ANYARRAY,
  INITCOND = '{}'
);

SELECT product_id, MAX(date_reported) AS latest
  ARRAY_TO_STRING(GROUP_ARRAY(bug_id), ',') AS bug_id_list,
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

另一些数据库并不支持自定义函数，因此实现这个解决方案需要写存储过程遍历一个非分组的查询结果，手动地将每个值连在一起。

当你希望非 Group By 列在每一组中只有一个值，但实际存储的数据依旧违背单值规则的情况下，可以使用这种方案。

遵循单值规则，避免获得模棱两可的查询结果。

第 16 章 随机选择

随机数的产生实在太重要了，不能够让它由偶然性来决定。

▶ 罗伯特·科维欧

一个带广告系统的网站，需要在用户每次访问页面的时候随机选择一个广告来展示，让每一个广告投递商都有同等机会展示其广告，并且读者也不会因为每次都显示一样的广告而觉得无聊。

最开始的几天一切都好，但是逐渐地，网站变得越来越慢。几个星期以后，人们就开始抱怨网站太慢了。你通过测试真实页面的加载速度发现，这并不是心理作用。你的读者开始失去兴趣，网站流量也在降低。

根据过往的经验，你首先想到使用性能工具和一个带有示例数据的测试数据库来定位性能瓶颈。但是奇怪的是，通过测试页面加载速度，发现用来生成页面的每一个 SQL 查询看上去都没有问题。但生产环境上的网站的确正变得越来越慢。

最终，你意识到生产环境上的数据库要比你的测试样本大得多得多。于是你用了一个同等规模的数据库重新进行测试，发现问题出在广告选择的那个查询上。随着广告的数量越来越多，随机选择的性能直线下降。你已经发现了这个扩展性很差的查询，这就迈出了重要的第一步。

你要怎么在网站丢失所有的读者和赞助商之前，重新设计这个随机选择广告的查询?

16.1 目标：获取样本记录

执行随机返回结果的查询的频率远大于我们的预期。从一个大的数据集中返回数据样例是很平常的事情，这似乎和可重用、确定性的程序设计原则相违背。比如下面这些情况：

- 轮流展示的内容，比如广告或者推荐的新闻；
- 审核记录的子集；

- 给当前可用的操作对象指派输入请求；
- 生成测试数据。

相比于将整个数据集读入程序中再取出样例数据集，直接通过数据库查询拿出这些样例数据集会更好。

本章的目标就是要写出一个仅返回随机数据样本的高效 SQL 查询[①]。

16.2 反模式：随机排序

获取随机记录的最常见 SQL 方法，就是对查询结果进行随机排序，然后获取第一行。这种技术理解起来很方便，实现起来也很方便：

Random/anti/orderby-rand.sql

```
SELECT * FROM Bugs ORDER BY RAND() LIMIT 1;
```

尽管这是一个很流行的解决方案，但它的弱点也很鲜明。要理解这种方案的弱点，我们要先来和普通排序进行一下比较。一般的排序过程中，我们对某一列中的值进行两两比较，根据比较结果来排序行。这种排序方法是可复用的，当你执行多次这样的排序时，每次返回的结果都是一样的。同时这种方式也能受益于索引，因为索引本身就是根据给定的列排序的。

Random/anti/indexed-sort.sql

```
SELECT * FROM Bugs ORDER BY date_reported;
```

如果排序的依据是给每一条记录分配一个随机值的函数，通过比较这个随机值来确定谁大谁小，那这样的结果就和记录本身的值无关，这样的排序每次执行的结果也都不一样。这正是目前我们想要的结果。

使用不定表达式（RAND()）意味着整个排序过程无法利用索引，因为没有索引会基于随机函数返回的值。这就是随机的作用：每次选择的时候都不同并且不可预测。

这就是影响查询性能的问题所在，因为使用索引是加速排序的最好方法。不使用索引的后果就是查询结果不得不由数据库“手动地”重新排序，这被称为一次*全表遍历*，并且经常伴随着将整个结果集保存到临时表中以及通过物理交换表内数据顺序操作来进行排序的情况。一次全表遍历排序比使用索引排序要慢很多，并且性能的差异随着数据量的增长而更加显著。

随机排序的另一个问题是，好不容易对整个数据集完成排序，但绝大多数的结果都浪费了，因为除了返回第一行之外，其他结果都立刻被丢弃了。在一个上千行的表中，为什么要费力地随机排序所有的记录，而我们所需要的仅仅是其中的一行？

---

① 数学家和计算机科学家对真实随机和伪随机进行了详细的区分。在实际中，计算机只能产生伪随机数。

这些问题在数据量很小的时候都是不容易被发现的，因此在开发和测试过程中它都可能是一个很好的方案。但随着数据量不断地增长，这个查询却无法很好地扩展。

16.3 如何识别反模式

在 16.2 节中描述的这个技术非常简单，并且很多程序员可能是在某篇文章中看到过或者自己研究过，都在使用这项技术。下面的这些问题可能就是你的同事用了本章的反模式的线索。

- "在 SQL 中，返回一个随机行真慢啊！"
  在开发和测试环境中，由于数据量较小，获取随机样本的查询语句工作得很好。但随着真实数据的增长，它也逐渐变得缓慢。没什么数据库调优方案、索引或者缓存能够提升它的扩展性。
- "我要怎么增加我的程序的可使用内存大小？我要获取所有的记录然后随机选择一个。"
  你不应该将所有数据载入到程序内存中，这么做非常浪费资源。除此之外，数据库的数据会不断增长直到程序内存完全不够用。
- "你是不是也觉得有些列出现的频率比别的要高一些？这个随机算法貌似不是很随机。"
  数据主键并不连续，并且随机数生成算法并没有考虑到这一点（参考 16.5 节）。

16.4 合理使用反模式

随机排序的性能问题在数据量很小的时候是可容忍的。

比如，你可以随机选择程序员去处理一个指定的 Bug。我们可以保证，程序员的数量永远不会大到需要使用高扩展性随机算法。

另一个例子就是从美国 50 个州中随机选择一个，这个列表的大小适度，而且在我们有生之年不太可能改变。

16.5 解决方案：没有具体的顺序……

随机选择是需要全表遍历并且耗时地进行手动排序的一个典型案例。在你设计 SQL 查询时，应该仔细地检查是否也有类似的查询。与其徒劳地想办法优化一个不可被优化的查询，还不如考虑别的实现方案。你可以使用接下来几段中所介绍的方法来获得一条随机记录。与随机排序不同的是，下面的每一个方案都能高效地获得同样的随机效果。

16.5.1 从 1 到最大值之间随机选择

一种避免对所有数据进行排序的方法，就是在 1 到最大的主键值之间随机选择一个。

Random/soln/rand-1-to-max.sql

```
SELECT b1.*
FROM Bugs AS b1
JOIN (SELECT CEIL(RAND() * (SELECT MAX(bug_id) FROM Bugs)) AS rand_id) AS b2
  ON (b1.bug_id = b2.rand_id);
```

这个方案假设主键的值是从 1 开始并且保持连续。这意味着在 1 到最大值之间没有任何值是未使用的。如果当中漏掉一些值，那随机获得的主键可能取不到任何数据。

当确信主键是从 1 到最大值连续的时候，可以使用这个方案。

16.5.2 选择下一个最大值

这个方案和前一个方案类似，但解决了在 1 到最大值之间有缝隙的情况，这个查询会返回它随机找到的第一个有效的值。

Random/soln/next-higher.sql

```
SELECT b1.*
FROM Bugs AS b1
JOIN (SELECT CEIL(RAND() * (SELECT MAX(bug_id) FROM Bugs)) AS bug_id) AS b2
WHERE b1.bug_id >= b2.bug_id
ORDER BY b1.bug_id
LIMIT 1;
```

这个方法解决了随机数没有主键的情况，同时也意味着在一个缝隙之后的那个值被选中的概率会增大。即使在一个离散的队列中，随机也应该保证每个值出现的概率相等，但这里的 bug_id 并不是如此。

当队列中的缝隙不大并且每个值要被等概率选择的重要性不高时，可以考虑使用这种方案。

16.5.3 获取所有的键值，随机选择一个

你可以使用程序代码来获取所有的主键值，然后随机选择一个。再使用这个随机选择出来的主键查询完整的记录。可以用如下的 PHP 代码实现：

Random/soln/rand-key-from-list.php

```
<?php
$bug_id_list = $pdo->query("SELECT bug_id FROM Bugs")->fetchAll();

$rand = random( count($bug_id_list) );
$rand_bug_id = $bug_id_list[$rand]["bug_id"];
$stmt = $pdo->prepare("SELECT * FROM Bugs WHERE bug_id = ?");
$stmt->execute( array($rand_bug_id) );
$rand_bug = $stmt->fetch();
```

这个方法避免了对全表的排序，并且选择每个键的概率相同，但它也有其他的开销。

- 获取所有的 bug_id 时会得到一个过长的列表，可能会超出程序所能处理的内存极限，并且导致如下的错误：

  ```
  Fatal error: Allowed memory size of 16777216 bytes exhausted
  ```

- 查询必须执行两次：一次获取主键的列表，第二次获取对应的记录。如果查询太复杂或者太耗时，这就会成为问题。

当查询逻辑很简单并且数据量适度的时候，可以考虑使用这个方案。这个方案在处理非连续值时效果很好。

16.5.4 使用偏移量选择随机行

还有另一个方案来避免之前几个方案中的问题，那就是计算总的数据行数，随机选择 0 到总行数之间的一个值，然后用这个值作为位移来获取随机行。

Random/soln/limit-offset.php

```
<?php

$rand = "SELECT ROUND(RAND() * (SELECT COUNT(*) FROM Bugs))";
$offset = $pdo->query($rand)->fetch(PDO::FETCH_ASSOC);

$sql = "SELECT * FROM Bugs LIMIT 1 OFFSET :offset";
$stmt = $pdo->prepare($sql);
$stmt->execute( $offset );
$rand_bug = $stmt->fetch();
```

这个方案使用了非标准的 LIMIT 子句，MySQL、PostgreSQL 和 SQLite 支持这一子句。

Oracle、Microsoft SQL Server 和 IBM DB2 使用另一个叫做 ROW_NUMBER()的函数。

比如，下面是 Oracle 的解决方案：

Random/soln/row_number.php

```
<?php
$rand = "SELECT 1 + MOD(ABS(dbms_random.random()),
  (SELECT COUNT(*) FROM Bugs)) AS offset FROM dual";
$offset = $pdo->query($rand)->fetch(PDO::FETCH_ASSOC);

$sql = "WITH NumberedBugs AS (
  SELECT b.*, ROW_NUMBER() OVER (ORDER BY bug_id) AS RN FROM Bugs b
) SELECT * FROM NumberedBugs WHERE RN = :offset";
$stmt = $pdo->prepare($sql);
$stmt->execute( $offset );
$rand_bug = $stmt->fetch();
```

当你不能保证主键是连续的，并且需要每行都有相同的选中概率时，可以用这个方案。

16.5.5　专有解决方案

每种数据库都可能针对这个需求提供独有的解决方案，比如 Microsoft SQL Server 2005 增加了一个 TABLE-SAMPLE 子句：

Random/soln/tablesample-sql2005.sql

```
SELECT * FROM Bugs TABLESAMPLE (1 ROWS);
```

Oracle 使用了一个类似的 SAMPLE 子句，比如返回表中 1%的记录：

Random/soln/sample-oracle.sql

```
SELECT * FROM (SELECT * FROM Bugs SAMPLE (1)
ORDER BY dbms_random.value) WHERE ROWNUM = 1;
```

你应该仔细阅读所使用的数据库的说明文档，来找这些专有解决方案。通常都有一些限制或者额外参数需要学习。

有些查询是无法优化的，换种方法试试看。

第 17 章

可怜人的搜索引擎

> 有些人遇到问题时总是会说：“我知道，我会使用正则表达式。”然后他会碰到更多问题。
>
> ▶ 杰米·加文斯基

1995 年，我在一家公司做技术支持，那时公司开始通过 Web 向客户提供信息。我们有一系列简短文档描述了常见问题的解决方案，我们想把这些文章放到 Web 做成一个知识库。

我们很快就意识到，随着文章数量的增长，知识库需要支持搜索功能，因为客户不想浏览几百篇文章才找到他们需要的解决方案。一个临时的应对方案是将文章分类，但即使如此，每个分类下的文章数量也很大，并且很多文章都可能属于多个分类。

我们想要让客户能够通过搜索文章，缩小候选列表。最灵活和直接的界面就是允许客户输入任意的关键字，然后给出包含这些关键字的文章。一篇文章如果和用户输入的关键字匹配度越高，则出现的越靠前。同时，我们也希望能够匹配词形变化。比如，对 crash 的搜索也要同时匹配 crashed、crashes 和 crashing。当然这个搜索要能够在一个不断增长的文章集合中快速地返回结果，这样才能在一个 Web 程序中使用它。

如果上面这些细节描述让对你觉得多余，我并不感到惊讶。搜索技术在如今已经变得如此平常，以至于我们都回想不起来它出现之前的情况了。但是用 SQL 搜索关键字，同时保证快速和精确，依旧是相当地困难。

17.1 目标：全文搜索

任何存储文本的应用都有针对这个文本进行单词或词组搜索的需求。我们使用数据库存储越来越多的文本数据，同时也需要搜索速度越来越快。Web 应用尤其需要高性能和高扩展性数据库搜索技术。

SQL 的一个基本原理（以及 SQL 所继承的关系原理）就是一列中的单个数据是*原子性的*。也就是说，你能对两个值进行比较，但通常是把两个值当成两个整体来比较。在 SQL 中比较子字符串总是意味着低效和不精确。

尽管如此，我们仍旧需要一个方法来比较一个较短的字符串和一个较长的字符串，并且查看是否较短字符串是较长字符串的一个子串。我们要怎么使用 SQL 来实现这样的需求呢？

17.2 反模式：模式匹配断言

SQL 提供了模式匹配断言来比较字符串，并且这是很多程序员用来搜索关键字的第一选择。最广泛支持的就是 LIKE 断言。

LIKE 断言提供了一个通配符（%）用以匹配 0 个或者多个字符。在一个关键字之前及之后使用通配符能够匹配到包含这个关键字的任意字符串。第一个通配符匹配了关键字之前的所有文本，第二个通配符匹配了关键字之后的所有文本。

Search/anti/like.sql

```
SELECT * FROM Bugs WHERE description LIKE '%crash%';
```

正则表达式也被很多数据库所支持，尽管这不是标准。你不需要使用通配符，因为基本上正则表达式能对所有子串进行模式匹配。如下是一个使用 MySQL 的正则表达式的例子[①]：

Search/anti/regexp.sql

```
SELECT * FROM Bugs WHERE description REGEXP 'crash';
```

使用模式匹配操作符的最大缺点就在于性能问题。它们无法从传统的索引上受益，因此必须进行全表遍历。由于对一个字符串列进行匹配操作非常耗时（相对来说，和比较两个整数是否相等所耗的时间相比），全表遍历所花的总时间就非常多。

使用 LIKE 或者正则表达式进行模式匹配搜索的另一个问题就是，经常会返回意料之外的结果。

Search/anti/like-false-match.sql

```
SELECT * FROM Bugs WHERE description LIKE '%one%';
```

上面的这个例子是要匹配包含 one 这个单词的文本，同时也匹配到单词 money、prone、lonely 等。在关键字两端都加上空格也不能完美解决这个问题，因为无法匹配到后面直接跟着标点符号的单词或者正好在文本开头或结尾的单词。数据库支持的正则表达式可能会为*单词边界*提供一个

① 尽管 SQL-99 标准定义了 SIMILAR TO 这个断言用来匹配正则表达式，但大多数 SQL 数据库还是使用非标准的语法。

模式来解决单词匹配问题[①]：

Search/anti/regexp-word.sql

```
SELECT * FROM Bugs WHERE description REGEXP '[[:<:]]one[[:>:]]';
```

考虑到性能和扩展性的问题，以及预防不合理的匹配所做的工作，简单的模式匹配对于关键字搜索来说是一个糟糕的技术方案。

17.3　如何识别反模式

如下的一些问题通常预示着项目中使用了“可怜人的搜索引擎”这个反模式。

- “我要怎么在 LIKE 表达式的两个通配符之间插入一个变量？”
  通常当程序员想要使用模式匹配对用户输入的关键字进行搜索时，会产生这个问题。
- “我要怎么写一个正则表达式来检查一个字符串是否包含多个单词、不包含一个特定的单词，或者包含给定单词的任意形式？”
  如果一个复杂的问题看起来用正则表达式很难解决，那就是用了反模式了。
- “我们网站的搜索功能在加了很多文档进去之后慢得不可理喻了。出什么问题了？”
  随着数据的增长，这个反模式方案就暴露出了它脆弱的扩展性问题。

17.4　合理使用反模式

在 17.2 节中所使用的 SQL 语句都是合法的，并且很简单直接。这意味着很多东西。

性能总是非常重要的，但一些查询过程很少执行，因此不需要花很多功夫对它进行优化。为一个很少使用的查询维护一个索引，可能就和用不高效的方法执行查询一样耗资源。如果这个查询是一个临时的查询，没人能保证你定义的索引能够给这个查询带来任何好处。

使用模式匹配操作进行复杂查询是很困难的，但如果你是为了一些简单的需求设计这样的模式匹配，它们就能帮助你用最少的工作量获得正确的结果。

17.5　解决方案：使用正确的工具

最好的方案就是使用特殊的搜索引擎技术，而不是 SQL。另一个可选方案是将结果保存起来从而减少重复的搜索开销。

接下来的几节介绍了一些数据库的内置扩展和独立项目所提供的搜索技术。同时，我们也会设计一个完全使用 SQL 标准的解决方案，它显然会比使用子串匹配更高效。

① 本例使用的是 MySQL 的语法。

17.5.1 数据库扩展

每个大品牌的数据库都有对全文搜索这个需求的解决方案，但这些方案并没有任何的标准，各个数据库的实现也互不兼容。如果只使用一种数据库品牌（或者打算使用开发商开发的特性），这些特性是获得高性能文本搜索的最佳途径，并且能和 SQL 查询整合得非常好。

如下是对一些 SQL 数据库的全文搜索技术的简要描述。具体的细节不断随数据库的版本升级而改变，因此记得要阅读一下对应当前使用版本的文档。

MySQL中的全文索引

MySQL 为 MyISAM 存储引擎提供了一个简单地全文索引类型。你可以在一个类型为 CHAR、VARCHAR 或者 TEXT 的列上定义一个全文索引。下面这个例子在 Bug 的 summary 和 description 列上定义了全文索引：

Search/soln/mysql/alter-table.sql

```
ALTER TABLE Bugs ADD FULLTEXT INDEX bugfts (summary, description);
```

可以使用 MATCH()函数对索引内容进行搜索。必须在匹配时指定需要全文索引的列（因而你可以在同一张表中对其他列使用不同类型的索引）。

Search/soln/mysql/match.sql

```
SELECT * FROM Bugs WHERE MATCH(summary, description) AGAINST ('crash');
```

自 MySQL 4.1 开始，你还能在表达式上使用简单的布尔运算来更精确地过滤查询结果。

Search/soln/mysql/match-boolean.sql

```
SELECT * FROM Bugs WHERE MATCH(summary, description)
  AGAINST ('+crash -save' IN BOOLEAN MODE);
```

Oracle中的文本索引

从 1997 年的 Oracle 8 开始，Oracle 就支持了文本索引特性，当时它是数据资料夹的一部分，称为 ConText。这项技术在随后的版本中多次更新，现在已经被集成到数据库程序里了。Oracle 中的文本索引技术复杂且强大，因此这里只能非常简单地总结一下。

- CONTEXT

  为单个文本列建立一个这样的索引类型，使用 CONTAINS()操作符进行搜索。索引和数据不保持同步，因此你每次都得手动重建索引或者定期重建。

  Search/soln/oracle/create-index.sql

  ```
  CREATE INDEX BugsText ON Bugs(summary) INDEXTYPE IS CTSSYS.CONTEXT;

  SELECT * FROM Bugs WHERE CONTAINS(summary, 'crash') > 0;
  ```

- CTXCAT

  这个类型的索引是针对短文本设计的，比如用在在线程序的分类目录上，和同一张表的其他结构化列一起。这种类型的索引会保持数据的同步。

  Search/soln/oracle/ctxcat-create.sql

```
CTX_DDL.CREATE_INDEX_SET('BugsCatalogSet');
CTX_DDL.ADD_INDEX('BugsCatalogSet', 'status');
CTX_DDL.ADD_INDEX('BugsCatalogSet', 'priority');

CREATE INDEX BugsCatalog ON Bugs(summary) INDEXTYPE IS CTSSYS.CTXCAT
  PARAMETERS('BugsCatalogSet');
```

  CATSEARCH()操作符分别接受两个参数来搜索索引列和结构化列的集合。

  Search/soln/oracle/ctxcat-search.sql

```
SELECT * FROM Bugs
WHERE CATSEARCH(summary, '(crash save)', 'status = "NEW"') > 0;
```

- CTXXPATH

  这种类型的索引是针对 XML 文档搜索而设计的，使用 existsNode()操作符。

  Search/soln/oracle/ctxxpath.sql

```
CREATE INDEX BugTestXml ON Bugs(testoutput) INDEXTYPE IS CTSSYS.CTXXPATH;

SELECT * FROM Bugs
WHERE testoutput.existsNode('/testsuite/test[@status="fail"]') > 0;
```

- CTXRULE

  假设在数据库中存了大量的文档，然后需要根据文档内容进行分类。

  使用 CTXRULE 索引，你可以设计分析文档的规则并返回文档的分类信息。同时，你可以提供一些示例文档以及对应的分类概念，然后让 Oracle 去分析这个规则并应用到其他文档上去。你甚至可以完全让这个过程自动化，让 Oracle 去分析文档结构然后自动分类。

  如何使用 CTXRULE 不在本书的讨论范围内。

Microsoft SQL Server的全文搜索

SQL Server 2000 及后续版本支持全文索引，它的配置相对复杂，包括了语言、同义词库和自动数据同步选项。SQL Server 提供了一系列的存储过程来创建全文索引，可以使用 CONTAINS()操作符来使用全文索引。

要执行对包含 crash 的 Bug 进行搜索的例子，首先要做的就是启用全文特性，然后在数据库中定义一个目录：

Search/soln/microsoft/catalog.sql

```
EXEC sp_fulltext_database 'enable'
EXEC sp_fulltext_catalog 'BugsCatalog', 'create'
```

接着，为 Bugs 表定义一个全文索引，将列加入到索引中，然后激活索引：

Search/soln/microsoft/create-index.sql

```
EXEC sp_fulltext_table 'Bugs', 'create', 'BugsCatalog', 'bug_id'
EXEC sp_fulltext_column 'Bugs', 'summary', 'add', '2057'
EXEC sp_fulltext_column 'Bugs', 'description', 'add', '2057'
EXEC sp_fulltext_table 'Bugs', 'activate'
```

启用全文索引的自动同步功能，从而对索引列的操作会同时影响到索引，然后就可以开始填充索引。这会在后台执行，因而需要花点时间才能使用索引。

Search/soln/microsoft/start.sql

```
EXEC sp_fulltext_table 'Bugs', 'start_change_tracking'
EXEC sp_fulltext_table 'Bugs', 'start_background_updateindex'
EXEC sp_fulltext_table 'Bugs', 'start_full'
```

最后，使用 CONTAINS()操作符执行查询：

Search/soln/microsoft/search.sql

```
SELECT * FROM Bugs WHERE CONTAINS(summary, '"crash"');
```

PostgreSQL的文本搜索

PostgreSQL 8.3 提供了一个复杂的可大量配置的方式，来将文本转化为可搜索的词汇集合，并且让这些文档能够进行模式匹配搜索。

为了最大地提升性能，你需要将内容存两份：一份为原始文本格式，另一份为特殊的 TSVECTOR 类型的可搜索格式。

Search/soln/postgresql/create-table.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  summary      VARCHAR(80),
  description  TEXT,
  ts_bugtext   TSVECTOR
  -- other columns
);
```

你需要确保 TSVECTOR 列的内容和你所想要搜索的列的内容同步。PostgreSQL 提供了一个内置的触发器来简化这一操作：

Search/soln/postgresql/trigger.sql

```
CREATE TRIGGER ts_bugtext BEFORE INSERT OR UPDATE ON Bugs
```

```
FOR EACH ROW EXECUTE PROCEDURE
  tsvector_update_trigger(ts_bugtext, 'pg_catalog.english', summary, description);
```

你也应该同时在 TSVECTOR 列上创建一个*反向索引*（GIN）：

Search/soln/postgresql/create-index.sql

```
CREATE INDEX bugs_ts ON Bugs USING GIN(ts_bugtext);
```

在做完这一切之后，就可以在全文索引的帮助下使用 PostgreSQL 的文本搜索操作符@@来高效地执行搜索查询：

Search/soln/postgresql/search.sql

```
SELECT * FROM Bugs WHERE ts_bugtext @@ to_tsquery('crash');
```

有很多其他的选项来自定义可搜索的内容、搜索查询和搜索结果。

SQLite的全文搜索（FTS）

SQLite 中的标准表结构并不支持高效的全文搜索，但你可以使用 SQLite 的一个可选扩展组件来存储可搜索的文本。这些内容会存储在一个特殊的*虚拟表*结构中。它有三个版本的扩展，FTS1、FTS2 和 FTS3。

FTS 扩展在标准的 SQLite 编译版本中默认是未启用的，因此你需要修改编译选项来启用 FTS，并重新编译 SQLite。比如，在 Makefile.in 中增加如下内容，然后编译 SQLite。

Search/soln/sqlite/makefile.in

```
TCC += -DSQLITE_CORE=1
TCC += -DSQLITE_ENABLE_FTS3=1
```

一旦你有了一个启用 FTS 的 SQLite 版本，就可以创建一个虚拟表来存储可搜索文本。任何数据类型、约束或者其他列选项将在搜索时被忽略。

Search/soln/sqlite/create-table.sql

```
CREATE VIRTUAL TABLE BugsText USING fts3(summary, description);
```

如果你在为另一张表建立索引(比如这个例子中的 Bugs 表),就必须将数据复制到虚拟表中。FTS 的虚拟表默认会包含一个主键列，叫做 docid，因此可以将原表中的行关联起来。

Search/soln/sqlite/insert.sql

```
INSERT INTO BugsText (docid, summary, description)
  SELECT bug_id, summary, description FROM Bugs;
```

现在你可以对 FTS 的虚拟表 BugsText 进行查询，使用一个高效的全文搜索断言 MATCH，然后把匹配的行和原表 Bugs 进行联结。将 FTS 表的名字当成一个虚拟的列来进行针对任意列的模式匹配。

Search/soln/sqlite/search.sql

```
SELECT b.* FROM BugsText t JOIN Bugs b ON (t.docid = b.bug_id)
WHERE BugsText MATCH 'crash';
```

匹配条件同时也支持功能有限的布尔表达式。

Search/soln/sqlite/search-boolean.sql

```
SELECT * FROM BugsText WHERE BugsText MATCH 'crash -save';
```

17.5.2　第三方搜索引擎

如果需要搜索功能的代码对不同的数据库都通用，那就需要一个独立于 SQL 数据库的搜索引擎。这一节简要介绍两个产品：Sphinx Search 和 Apache Lucene。

Sphinx Search

Sphinx Search（http://www.sphinxsearch.com/）是一个开源的搜索引擎，用于和 MySQL 及 PostgreSQL 配套使用。在本书出版时，一个非官方的 Sphinx Search 补丁支持开源的 Firebird 数据库。可能在将来的版本中，这个搜索引擎会考虑支持其他的数据库。

在 Sphinx Search 中，构建索引和搜索都很快，而且它也支持分布式查询。对于数据不常更新且要求高可扩展性的程序来说，Sphinx Search 是个很好的选择。

你可以使用 Sphinx Search 来索引存在于 MySQL 数据库中的数据。通过修改 `sphinx.conf` 中的几个字段，你可以指定对应的数据库。你必须写一些 SQL 查询脚本为构建索引的操作获取数据。这个查询要求第一列为一个整型主键。你可以声明一些属性列来对结果进行约束或者排序。余下的列就是用来进行全文索引的。最后，另一个 SQL 查询脚本通过给定的主键代码`$id`，从数据库中获取完整的记录。

Search/soln/sphinx/sphinx.conf

```
source bugsrc
{
  type                 = mysql
  sql_user             = bugsuser
  sql_pass             = xyzzy
  sql_db               = bugsdatabase
  sql_query            = \
    SELECT bug_id, status, date_reported, summary, description \
    FROM Bugs
  sql_attr_timestamp   = date_reported
  sql_attr_str2ordinal = status
  sql_query_info       = SELECT * FROM Bugs WHERE bug_id = $id
}

index bugs
```

```
{
  source                = bugsrc
  path                  = /opt/local/var/db/sphinx/bugs
}
```

在配置完成 `sphinx.conf` 之后，你就可以在 shell 中使用 `indexer` 命令创建索引了：

Search/soln/sphinx/indexer.sh

```
indexer -c sphinx.conf bugs
```

你可以使用 `search` 命令对索引进行搜索：

Search/soln/sphinx/search.sh

```
search -b "crash -save"
```

Sphinx Search 也有一个守护进程以及对应的 API 给常用的脚本语言（诸如 PHP、Perl 和 Ruby）使用。当前版本的最主要问题是，目前的索引算法不支持高效的增量更新。在一个经常更新的数据源上使用 Sphinx Search 需要一些折中的处理办法。比如说，将可搜索表拆分成两张表，第一张表存储不变的主体历史数据，第二张表存储一个较小的当前数据的集合。当数据逐渐增长的时候，就需要重建索引。然后你的程序必须对两个 Sphinx Search 索引进行搜索。

Apache Lucene

Lucene（http://lucene.apache.org/）是一个针对 Java 程序的成熟搜索引擎。还有一些类似的项目，只是所使用的语言不同，包括 C++、C#、Perl、Python、Ruby 和 PHP。

Lucene 使用其独有的格式为文本文档创建索引。Lucene 索引不和元数据保持同步。如果你插入、删除或者更新数据库中的数据，必须同时也对 Lucene 索引进行对应的操作。

使用 Lucene 搜索引擎有点像使用一台汽车发动机，必须要有一堆相关技术支持才能让它工作。Lucene 不会直接从 SQL 数据库中读取数据集合，你必须手动往 Lucene 索引中写入文档。比如，你可以执行一个 SQL 查询，然后对结果中的每一行，创建一个 Lucene 文档对象然后保存到 Lucene 索引中。你可以通过 Lucene 的 Java API 来使用对应的功能。

所幸的是，Apache 提供了另一个项目叫做 Solr（http://lucene.apache.org/solr/）。Solr 是一个 Lucene 索引的网关服务。你可以向 Solr 添加文档或者使用一个 REST 风格的接口提交查询请求，然后就可以使用任意的语言来调用 Lucene 的服务了。

你也可以将 Solr 配置成直接连接到数据库，执行一个查询操作，然后使用 `DataImportHandler` 工具来对结果进行索引。

实现自己的搜索引擎

假设你不想使用不同数据库特有的搜索特性，也不想安装一个独立的搜索引擎产品。你需

要一个高效的、与数据库品牌无关的解决方案来进行文本搜索。在本节中，我们将使用一个称为反向索引的方案。基本上来说，反向索引就是一个所有可能被搜索的单词列表。在多对多的关系中，索引将这些单词和包含这些单词的文本关联起来。也就是说，crash 这个单词出现在很多 Bug 描述中，然后每个 Bug 描述又包含很多别的关键字。这一节就来介绍如何设计反向索引。

首先，定义一张 Keywords 表来记录所有用户用来搜索的关键字，然后定义个交叉表 BugsKewords 来建立多对多的关系：

Search/soln/inverted-index/create-table.sql

```
CREATE TABLE Keywords (
  keyword_id  SERIAL PRIMARY KEY,
  keyword     VARCHAR(40) NOT NULL,
  UNIQUE KEY (keyword)
);

CREATE TABLE BugsKeywords (
  keyword_id  BIGINT UNSIGNED NOT NULL,
  bug_id      BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (keyword_id, bug_id),
  FOREIGN KEY (keyword_id) REFERENCES Keywords(keyword_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

接下来，将每个关键字和其所匹配到的 Bug 添加到 BugsKeywords 表中。我们可以使用 LIKE 或者正则表达式来执行子字符串匹配查询，获得我们所需要的匹配记录。这种方式不比在 17.2 节中所说的查询有更多的开销，但由于我们只执行一次这样的查询，因此省下了很多时间。如果我们将查询结果存储在交叉表中，所有之后对同一个关键字的搜索都会快很多。

接下来，我们写一个存储过程来简化对一个给定关键字的搜索[①]。如果给定的关键字已经被搜索过了，这个查询就会很快，因为 BugsKeywords 表中的记录就是包含这个给定的关键字的文章列表。如果还没人搜索过这个给定的关键字，我们就需要使用原始的方法对所有的文本记录进行全文搜索。

Search/soln/inverted-index/search-proc.sql

```
CREATE PROCEDURE BugsSearch(keyword VARCHAR(40))
BEGIN
  SET @keyword = keyword;
❶     PREPARE s1 FROM 'SELECT MAX(keyword_id) INTO @k FROM Keywords
        WHERE keyword = ?';
      EXECUTE s1 USING @keyword;
      DEALLOCATE PREPARE s1;
```

① 这个例子使用 MySQL 的语法来编写存储过程。

```
    IF (@k IS NULL) THEN
❷     PREPARE s2 FROM 'INSERT INTO Keywords (keyword) VALUES (?)';
      EXECUTE s2 USING @keyword;
      DEALLOCATE PREPARE s2;
❸     SELECT LAST_INSERT_ID() INTO @k;

❹     PREPARE s3 FROM 'INSERT INTO BugsKeywords (bug_id, keyword_id)
        SELECT bug_id, ? FROM Bugs
        WHERE summary REGEXP CONCAT(''[[:<:]]'', ?, ''[[:>:]]'')
          OR description REGEXP CONCAT(''[[:<:]]'', ?, ''[[:>]]'')';
      EXECUTE s3 USING @k, @keyword, @keyword;
      DEALLOCATE PREPARE s3;
    END IF;

❺   PREPARE s4 FROM 'SELECT b.* FROM Bugs b
      JOIN BugsKeywords k USING (bug_id)
      WHERE k.keyword_id = ?';
    EXECUTE s4 USING @k;
    DEALLOCATE PREPARE s4;
  END
```

❶ 搜索用户指定关键字。从 keywords、keyword_id 或 NULL 返回整型主键，如果这个词之前未出现过。

❷ 如果未找到该词，将它当做新词插入。

❸ 查询 keywords 生成的主键值。

❹ 通过搜索有新关键字的 Bug 来填入交叉表。

❺ 最后，查询符合 keyword_id 的整行，无论这个关键字存在或者需要当做新词条插入。

现在我们可以执行这个存储过程，然后传入想要的关键字。这个存储过程会返回匹配到的 Bug，不论它是需要搜索全表来找到匹配的 Bug 然后追加到交叉表中，还是直接就可以从交叉表中获取相应的结果。

Search/soln/inverted-index/search-proc.sql

```
CALL BugsSearch('crash');
```

这个方案还有一个需要注意的是：在有新文档添加到数据库中时，我们需要定义一个触发器以填充交叉表。如果你需要支持对 Bug 描述的更新操作，还要写一个触发器去重新分析文本，然后添加或删除 BugsKeywords 表中的记录。

Search/soln/inverted-index/trigger.sql

```
CREATE TRIGGER Bugs_Insert AFTER INSERT ON Bugs
FOR EACH ROW
BEGIN
  INSERT INTO BugsKeywords (bug_id, keyword_id)
    SELECT NEW.bug_id, k.keyword_id FROM Keywords k
    WHERE NEW.description REGEXP CONCAT('[[:<:]]', k.keyword, '[[:>:]]')
```

```
        OR NEW.summary REGEXP CONCAT('[[:<:]]', k.keyword, '[[:>:]]');
END
```

这个关键字列表会随着用户不断地搜索而自动增长，因此我们不必将每个在知识库中找到的单词都加到列表中去。从另一个角度来说，如果我们可以预测一些关键字，就可以简单地在程序初始化的时候执行一下这几个关键字的搜索。这样，这部分的时间开销就不会由用户来承担了。

在本章最开始的知识库的例子中我使用了反向索引，我还在Keywords表中添加了额外的一列num_searches。这一列在每次这个关键字被搜索时会自增，我使用这一列来跟踪搜索关键字的分布。

你不必使用 SQL 来解决所有问题。

第 18 章

意大利面条式查询

实体不应不必要地增殖。

▶ 奥卡姆

你的老板正和他的老板打电话，然后他示意你过去。他用手遮住了话筒，小声对你说："执行委员会在开预算会议，我们可能要被裁员，除非我能有数据告诉副总裁我们的部门一直都很忙。我需要知道我们同时在开发多少个项目，多少个程序员在修补 Bug，每个程序员平均修补多少个 Bug，以及多少修复了的 Bug 是由用户报告的。现在就要！"

你飞奔至座位，打开 SQL 工具，开始写查询语句。你想要立刻得到所有的数据，于是你写了一个很复杂的查询脚本，希望能尽可能减少重复的工作量，能更快地得到数据。

Spaghetti-Query/intro/report.sql

```
SELECT COUNT(bp.product_id) AS how_many_products,
  COUNT(dev.account_id) AS how_many_developers,
  COUNT(b.bug_id)/COUNT(dev.account_id) AS avg_bugs_per_developer,
  COUNT(cust.account_id) AS how_many_customers
FROM Bugs b JOIN BugsProducts bp ON (b.bug_id = bp.bug_id)
JOIN Accounts dev ON (b.assigned_to = dev.account_id)
JOIN Accounts cust ON (b.reported_by = cust.account_id)
WHERE cust.email NOT LIKE '%@example.com'
GROUP BY bp.product_id;
```

数据出来了，但看上去是错的。我们怎么会有几十个产品？怎么可能平均每个程序员正好修复 1.0 个 Bug？你的老板要的不是客户数量，他要得是客户报告的 Bug 数量。这些数据怎么会是这样呢？这个查询比你所想的要更加复杂。

你的老板挂掉了电话。"无所谓了，"他叹息道，"太晚了。我们收拾下桌子吧。"

18.1 目标：减少 SQL 查询数量

SQL 开发人员经常会被同一个问题困扰："我要怎么用一个查询来完成这件事情？"这个问

题基本上在任务中都会被提及。受过培训的程序员认为，一个 SQL 查询是困难的、复杂的和耗资源的，那么两个 SQL 查询就是糟糕度乘以二。用多于两个的 SQL 查询来解决问题根本不在考虑范围内。

程序员不能减少他们任务的复杂度，但他们想要简单化其解决方案。他们使用“优雅”或者“高效”这些形容词来描述他们的目标，并且认为使用一条 SQL 查询就能完成目标。

18.2 反模式：使用一步操作解决复杂问题

SQL 是一门极具表现力的语言——你可以在单个 SQL 查询或者单条语句中完成很多事情。但这并不意味着必须强制只使用一行代码，或者认为使用一行代码就搞定每个任务是个好主意。你在使用其他语言的时候也有这样的习惯吗？应该没有吧。

18.2.1 副作用

通过一个查询来获得所有结果的常见后果就是得到了一个笛卡儿积。当查询中的两张表之间没有条件限制其关系时，就会发生这样的情况。没有对应的限制而直接使用两张表进行联结查询，就会得到第一张表中的每一行和第二张表中的每一行的一个组合。每一个这样的组合就会成为结果集中的一行，最终你就得到一个行数多很多的结果集。

我们来看个例子。假设我们想要查询 Bugs 数据库，计算一个给定的产品有多少 Bug 被修复了，多少 Bug 正处于打开状态。很多程序员可能会写出如下的这条语句：

Spaghetti-Query/anti/cartesian.sql

```
SELECT p.product_id,
  COUNT(f.bug_id) AS count_fixed,
  COUNT(o.bug_id) AS count_open
FROM BugsProducts p
LEFT OUTER JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status = 'FIXED')
LEFT OUTER JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status = 'OPEN')
WHERE p.product_id = 1
GROUP BY p.product_id;
```

你碰巧知道，事实上对于给定的这个产品，有 12 个 Bug 被修复了，有 7 个 Bug 是打开的。因此，结果看上去很耐人寻味：

Product_id	count_fixed	count_open
1	84	84

是什么导致结果和预期相差十万八千里？没那么巧，正好是 $84 = 12 \times 7$。这个例子将 Products 表和两个不同的 Bugs 表的子集联合起来，结果却是那两个子集的笛卡儿积。12 个 FIXED 状态的 Bug 中的每一个和一个 OPEN 状态的 Bug 凑成了一对。

你可以想象，笛卡儿积和图 18-1 所画的一样。每条线链接了一个已修复的 Bug 和一个打开的 Bug，然后成为了临时结果集中的一行（在分组语句执行之前）。我们可以注释掉 GROUP BY 子句和那些聚合函数来查看这个查询的结果。

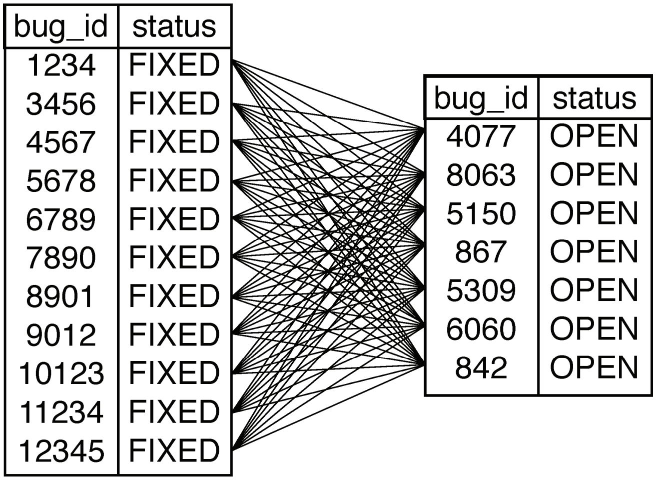


图 18-1 被修复与打开 Bug 的笛卡儿积

Spaghetti-Query/anti/cartesian-no-group.sql

```
SELECT p.product_id, f.bug_id AS fixed, o.bug_id AS open
FROM BugsProducts p
JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status = 'FIXED')
JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status = 'OPEN')
WHERE p.product_id = 1;
```

这个查询唯一描述的关系是 BugsProducts 表和每个 Bugs 子集之间的关系。没有条件来约束每个 FIXED 状态的 Bug 和每个 OPEN 状态的 Bug 是否能进行配对，而默认情况下它们是可以配对的。最终的结果是得到一个 12 乘以 7 的结果集。

当你尝试着执行一个类似的双重任务的查询时，很容易就得到一个意料之外的笛卡儿积。如果你尝试在一个查询中完成更多不相关的工作，最终的结果可能是在此基础上再多乘出一个笛卡儿积。

18.2.2 那好像还不够……

除了你会得到错误结果之外，这些查询也非常难写、难以修改和难以调试。数据库查询请求的日益增加应该是预料之中的事。经理们想要更复杂的报告以及在用户界面上添加更多的字段。如果你的设计很复杂，并且是一个单一查询，要扩展它们就会很费时费力。不论对你还是对项目来说，时间花在这些事情上面不值得。

此外，还有运行时开销。一条精心设计的复杂 SQL 查询，相比于那些直接简单的查询来说，不得不使用很多的 JOIN、关联子查询和其他让 SQL 引擎难以优化和快速执行的操作符。程序员直觉地认为越少的 SQL 执行次数性能越好，如果 SQL 查询的复杂度都相同时的确如此。但另一方面，一个怪兽般的 SQL 查询的开销可能成指数级别增长，而使用多个简单的查询却有更好的效果。

18.3　如何识别反模式

如果你听见项目组成员说了下面这些话，可能就是使用了“意大利面条式查询”这个反模式。

- “为什么我的求和、计数返回的结果异常地大？”
  一个意料之外的两个联结查询的数据集生成的笛卡儿积。
- “我一整天都在和这个变态的查询语句做斗争！”
  SQL 并不是那么难的，真的！如果你和单条 SQL 查询纠结了很长时间，应该重新考虑你的实现方式。
- “我们不能在数据库报表中再加入任何新东西了，因为对查询语句的修改要花很长时间。”
  写这个查询语句的人要永远为他所写的代码负责，即使他们已经加入到别的项目中去了。那个写这段代码的人可能就是你，因此别把查询写得如此复杂以至于别人无法维护！
- “试试看再加一个 DISTINCT 进去？”
  要修正由于笛卡儿积所带来的数据集暴涨，程序员通常使用 DISTINCT 这个关键字作为一个查询修正或者一个聚合函数来减少重复。这个方法能够隐藏掉那个难看的查询的踪迹，但导致 RDBMS 做了更多的工作来生成排序、去重的临时表。

另一个表明一个查询可能是意大利面条式查询的证据是它的执行时间很长。低劣的性能可能是其他问题的一个征兆，但你在调查时应该考虑到可能在某个 SQL 语句中做了太多事情。

18.4　合理使用反模式

需要将一个复杂的查询任务放在一个 SQL 查询中完成的最常见原因，是你正在使用一个编程框架或者一个可视化组件库直接和数据源相连，然后在程序中直接展示数据。简单的商务智能和报表工具都属于这一分类中，但大多数高级的 BI 软件可以从多个数据源合并数据。

组件或者报表工具通常假设单个 SQL 查询仅用来完成一个简单的任务，但它鼓励你去设计更庞大的查询来生成报告中的所有数据。如果你使用某个这样的报表程序，就可能被迫去写一个更复杂的 SQL 查询，而没有机会写代码操作结果集。

如果报表需求太复杂而不能用单个 SQL 查询完成，更好的方案可能是生成多个报表。如果你的老板不喜欢这样的解决方案，要提醒他报表的复杂度和生成报表所花的时间是成正比的。

有时候，你想要在一个 SQL 查询中得到一个复杂的结果，是因为需要所有的这些结果能排序后再组合在一起。在 SQL 查询中指定一个排序是很简单的。理论上来说让数据库来执行这个操作比你自己写代码实现多个请求结果的排序要高效得多。

18.5 解决方案：分而治之

奥卡姆[①]在本章开头的名句也称为简约律。

简约律

当你有两个相互竞争的理论能得出同样的结论，那么简单的那个更好。

对于 SQL 来说，那意味着你在两个能得到同样结果的查询之中做选择时，选择更简单的那个。我们在修正本章的反模式查询时，应时刻谨记这个定律。

18.5.1 一步一个脚印

如果你看不出，在意外产生了笛卡儿积的两张表间存在逻辑关联关系，那有一个很简单的解释：根本就没有这种关系。要避免生成这个笛卡儿积，你不得不将这个意大利面条式查询拆分成几个小而简单的查询。在之前描述的那个简单例子中，我们只需要两个查询：

Spaghetti-Query/soln/split-query.sql

```
SELECT p.product_id, COUNT(f.bug_id) AS count_fixed
FROM BugsProducts p
LEFT OUTER JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status = 'FIXED')
WHERE p.product_id = 1
GROUP BY p.product_id;

SELECT p.product_id, COUNT(o.bug_id) AS count_open
FROM BugsProducts p
LEFT OUTER JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status = 'OPEN')
WHERE p.product_id = 1
GROUP BY p.product_id;
```

这两个查询的结果分别是 12 和 7，正如我们所希望的。

你可能会觉得将一个查询拆分成多个查询是一个“不雅”的解决方案，并且略感懊屈和遗憾，但你很快就会意识到，在开发、维护和性能方面，这么做能带来一些积极的影响。

- 这个查询不会产生早先例子中出现的那种意外的笛卡儿积，因此很简单地就能确认这个查询给出的结果是精确的。

① William of Ockham，十四世纪英国著名思想家，逻辑学家。著有“奥卡姆剃刀”理论。——编者注

- 当有新的需求增加到报表中时，添加另一个简单的查询，比将更多的计算整合到一个已经很复杂的查询中去要简单很多。
- SQL 引擎能更容易和可靠地对简单的查询进行优化和执行。即使整个工作看上去像是被分割出来的查询弄得有点重复，但可能执行得更快。
- 在代码审查或者团队间的培训交流时，解释几个易懂的查询要比解释一个庞大复杂的查询容易得多。

18.5.2 寻找UNION标记

你可以将几个查询的结果进行 UNION 操作，从而最终得到一个结果集。当你确实想要提交单个查询并且得到单个结果集时，这么做很有帮助，比如在需要保存查询结果时。

Spaghetti-Query/soln/union.sql

```
(SELECT p.product_id, f.status, COUNT(f.bug_id) AS bug_count
 FROM BugsProducts p
 LEFT OUTER JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status = 'FIXED')
 WHERE p.product_id = 1
 GROUP BY p.product_id, f.status)

UNION ALL

(SELECT p.product_id, o.status, COUNT(o.bug_id) AS bug_count
 FROM BugsProducts p
 LEFT OUTER JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status = 'OPEN')
 WHERE p.product_id = 1
 GROUP BY p.product_id, o.status)

ORDER BY bug_count;
```

这个查询的结果是每个子查询结果联合后所得的。在本例中有两行，每行对应一个子查询。请记住要额外地增加一列来区分不同子查询的结果，本例中是 status 这一列。

仅在两个子查询的列属性是相互兼容的情况下才能使用 UNION。你不能在查询的中间改变列的数值、名字或者数据类型，因此需要确保所有行的所有列都是相同的。如果你发现定义了一个列的别名，类似于 bugcount_or_customerid_or_null，很有可能就是你对不兼容的两个结果集使用了 UNION。

18.5.3 解决老板的问题

怎么解决这个统计项目信息的紧急任务？你的老板说："我需要知道我们同时在开发多少个项目，多少个程序员在修补 Bug，每个程序员平均修补多少个 Bug，以及多少修复了的 Bug 是由用户报告的。"

做好的解决方案就是拆分所有的这些工作。

- 多少产品：

Spaghetti-Query/soln/count-products.sql

```
SELECT COUNT(*) AS how_many_products
FROM Products;
```

- 多少开发人员在参与修补 Bug：

Spaghetti-Query/soln/count-developers.sql

```
SELECT COUNT(DISTINCT assigned_to) AS how_many_developers
FROM Bugs
WHERE status = 'FIXED';
```

- 平均每个程序员修复了多少 Bug：

Spaghetti-Query/soln/bugs-per-developer.sql

```
SELECT AVG(bugs_per_developer) AS average_bugs_per_developer
FROM (SELECT dev.account_id, COUNT(*) AS bugs_per_developer
      FROM Bugs b JOIN Accounts dev
        ON (b.assigned_to = dev.account_id)
      WHERE b.status = 'FIXED'
      GROUP BY dev.account_id) t;
```

- 多少修复了的 Bug 是由客户报告的：

Spaghetti-Query/soln/bugs-by-customers.sql

```
SELECT COUNT(*) AS how_many_customer_bugs
FROM Bugs b JOIN Accounts cust ON (b.reported_by = cust.account_id)
WHERE b.status = 'FIXED' AND cust.email NOT LIKE '%@example.com';
```

其中的几个查询其本身就已经足够复杂了，要再将它们合并到单个输出结果集中，简直就是噩梦！

18.5.4 使用SQL自动生成SQL

当你拆分一个复杂的 SQL 查询时，得到的结果可能是很多相似的查询，可能仅仅在数据类型上面有所不同。编写所有的这些查询是很乏味的，因此，最好能够有个程序自动生成这些代码。

代码生成是一种输出一段新的可以编译或者执行的代码的写代码技术。如果手写这些代码很费力，代码生成技术就是非常有价值的。一个代码生成器可以帮你去除重复的工作。

多表更新

在做咨询时，我被叫去为另一个部门的经理解决一个紧急的 SQL 问题。

我走进了经理办公室，看到他穷途末路的苦恼样子。我们简单地互相问候了一下，他就开始向我解释他所面临的困境：“我希望你能快速地解决这个问题。我们的库存系统已经下线一整天了。”他并不是 SQL 的业余开发者，但他告诉我他已经在一个用来同时更新大量记录的 SQL 语句上花了好几个小时了。

他的问题是他没办法在 UPDATE 语句上对所有的值使用固定的 SQL 表达式。事实上，每一行上需要更新的值都是不一样的。他的数据库跟踪一个计算机中心的库存信息以及每台电脑的使用情况。他想要添加一个 last_used 列记录每台电脑的最后一次使用日期。

他太专注于使用单个 SQL 语句来解决这个复杂的问题了，这是另一个“意大利面条式查询”的例子！他这几个小时想要写出这个完美的 UPDATE 的时间，都可以手动更新掉所有的记录了。

Spaghetti-Query/soln/generate-update.sql

```
SELECT CONCAT('UPDATE Inventory '
  ' SET last_used = ''', MAX(u.usage_date), '''',
  ' WHERE inventory_id = ', u.inventory_id, ';') AS update_statement
FROM ComputerUsage u
GROUP BY u.inventory_id;
```

和他想要写出一个 SQL 语句来解决这个复杂问题不同，我写了一个脚本来生成一系列更简单且符合需求的 SQL 语句：

这个查询的输出是一系列的 UPDATE 语句，由分号分割，可以直接作为 SQL 脚本执行：

Update_statement
UPDATE Inventory SET last_used = '2002-04-19' WHERE inventory_id = 1234;
UPDATE Inventory SET last_used = '2002-03-12' WHERE inventory_id = 2345;
UPDATE Inventory SET last_used = '2002-04-30' WHERE inventory_id = 3456;
UPDATE Inventory SET last_used = '2002-04-04' WHERE inventory_id = 4567;

通过这种方法，我在几分钟内就解决了这个经理花了几小时在那里纠结的问题。

执行多次 SQL 查询或者多条 SQL 语句可能并不是解决问题最高效的办法，但你应该在效率和解决问题之间找到平衡点。

尽管 SQL 支持用一行代码解决复杂的问题，但也别做不切实际的事情。

第 19 章 隐式的列

连我自己都不知道自己要说什么，怎么告诉你我在想什么？

▶ E. M.福斯特

一个 PHP 开发人员向我寻求帮助，他的图书馆数据库在执行一个看上去很简单的 SQL 查询时的行为让人疑惑不解。

Implicit-Columns/intro/join-wildcard.sql

```
SELECT * FROM Books b JOIN Authors a ON (b.author_id = a.author_id);
```

这个查询返回的所有的书名都是 NULL。更奇怪的是，当他执行一个不带 Authors 表的查询时，返回的结果又是和预期的一样，包含了正确的书名。

我帮他找到了问题的根源：他所使用的 PHP 数据库扩展将从 SQL 查询返回的每条记录都表示成一个关联数组。比如，使用$row["isbn"]直接获得 Books.isbn 的值。在他所设计的表中，Books（书）和 Authors（作者）两张表里都有一个 title（有“标题”和“称呼”两个意思）列。由于结果数组中的单个条目$row["title"]仅能存储一个值，在这个例子中，Authors.title 占据了这个数组条目。而在数据库中，大多数作者的 title 这一列都没有值，因此$row["title"]的值就等于 NULL。当这个查询不包含 Accounts 表时，列名之间没有冲突，书名这一列就如同我们预期的那样占据了这个数组条目。

我告诉这个程序员，解决方案就是给其中的一个 title 声明一个别名，这样不同 title 就会使用数组中不同的条目。

Implicit-Columns/intro/join-alias.sql

```
SELECT b.title, a.title AS salutation
FROM Books b JOIN Authors a ON (b.author_id = a.author_id);
```

随后他问了我第二个问题：“我要怎么只给一个列定义别名，同时还要获取其余的所有列？”他想要继续使用通配符（SELECT *），又要给通配符所包含的某一列定义别名。

19.1 目标：减少输入

软件开发人员似乎不太愿意打字，这在某种程度上是对他们选择的这个职业的讽刺，就像欧亨利的小说中那对双胞胎的结局。

程序员通常用下面这个需要写出所有查询列的例子来说明要打的字太多了：

Implicit-Columns/obj/select-explicit.sql

```
SELECT bug_id, date_reported, summary, description, resolution,
  reported_by, assigned_to, verified_by, status, priority, hours
FROM Bugs;
```

程序员喜欢使用 SQL 通配符，我一点也不觉得惊讶。符号*意味着所有的列，因此列的列表是隐式定义的，而不是显式的。这让查询代码变得更清晰。

Implicit-Columns/obj/select-implicit.sql

```
SELECT * FROM Bugs;
```

同样地，当使用 INSERT 时，使用默认的方案似乎更好：输入的数据会按照列在表中定义的顺序应用到所有的列上。

Implicit-Columns/obj/insert-explicit.sql

```
INSERT INTO Accounts (account_name, first_name, last_name, email,
  password, portrait_image, hourly_rate) VALUES
  ('bkarwin', 'Bill', 'Karwin', 'bill@example.com', SHA2('xyzzy'), NULL, 49.95);
```

不需要列出列名让 SQL 语句变得更短。

Implicit-Columns/obj/insert-implicit.sql

```
INSERT INTO Accounts VALUES (DEFAULT,
  'bkarwin', 'Bill', 'Karwin', 'bill@example.com', SHA2('xyzzy'), NULL, 49.95);
```

19.2 反模式：捷径会让你迷失方向

尽管使用通配符和未命名列能够达到减少输入的目的，但这个习惯也会带来一些危害。

19.2.1 破坏代码重构

假设你要向 Bugs 表里增加一列，比如说用来安排日程的 date_due 列。

Implicit-Columns/anti/add-column.sql

```
ALTER TABLE Bugs ADD COLUMN date_due DATE;
```

INSERT 语句现在会报错，因为现在这张表需要 12 个传入参数，而你只有 11 个。

Implicit-Columns/anti/insert-mismatched.sql

```
INSERT INTO Bugs VALUES (DEFAULT, CURDATE(), 'New bug', 'Test T987 fails...',
    NULL, 123, NULL, NULL, DEFAULT, 'Medium', NULL);

-- SQLSTATE 21S01: Column count doesn't match value count at row 1
```

在使用隐式模式执行 INSERT 时，输入必须严格按照定义表时的那些列的顺序。如果列变了，这条语句就会抛出一个错误，甚至有可能把数据写到错误的列里面去。

假设你要执行一个 SELECT *的查询，由于不知道具体的列名，所以你只能按照最开始设计表的顺序来使用结果：

Implicit-Columns/anti/ordinal.php

```
<?php
$stmt = $pdo->query("SELECT * FROM Bugs WHERE bug_id = 1234");
$row = $stmt->fetch();
$hours = $row[10];
```

但是在你不知道的情况下，有人删掉了一列：

Implicit-Columns/anti/drop-column.sql

```
ALTER TABLE Bugs DROP COLUMN verified_by;
```

hours 这一列已经不是第十列了，于是程序错误地使用了另一列的数据。在重命名、添加、删除列的时候，程序代码并不能适应查询结果的改变。如果使用了通配符，就无法预测这个查询会返回多少行。

这些错误可能会隐藏得很深，当你在程序的输出中发现问题的时候，很难回溯并定位到出问题的那行代码。

19.2.2 隐藏的开销

在查询中使用通配符可能会影响性能和扩展性。一次查询所获取的列越多，客户端程序和数据库之间的网络传输的字节数也越多。

生产环境中的程序可能会有很多并发的查询请求。它们都共享同一个网络带宽，即使一个千兆网络环境也可能由于上百个客户端同时查询返回上千条记录而造成阻塞。

诸如 Active Record 这类的对象关系映射（ORM）技术通常默认使用 SELECT *取得的数据来填充一个表示数据库行的对象。即使 ORM 提供了一些修改默认行为方式的接口，大多数程序员也懒得去改。

19.2.3 你请求，你获得

我所遇到的程序员使用 SQL 通配符时问得最多的问题是：“有没有选择除了几个我不想要的

列之外所有列的方法？”可能这些程序员想要避免获取庞大的 TEXT 类型的列的开销，但他们同时也想要获得通配符所带来的书写上的便利。

答案是“没有”。SQL 不支持这种“除了我不想要的，其他都要”的语法。你只能使用通配符获取一张表的所有列，或者一个个显式地列出所有你想要的列。

19.3 如何识别反模式

如下的情形可能预示着你的项目在使用隐式列的时候处理得不恰当，并且造成了一定的麻烦。

- “程序由于还使用老的列名而挂掉了。我们尝试了更新所有相关的代码，但可能还有地方漏掉了。”
  你改变了数据库里的一张表——添加、删除、重命名列，或者改变列的顺序——但没能更新全部使用到这张表的代码。要找到所有对这张表的引用是件工作量很大的事情。
- “我们花了几天时间终于找到了网络的瓶颈，最终我们减小了到数据库服务器的庞大的通信量。根据我们的统计信息，平均每个查询请求获取 2MB 的数据，但只有十分之一是用来显示的。”
  你获取了一堆用不到的数据。

19.4 合理使用反模式

在你只是为了快速地写几个脚本对一个解决方案进行测试，或者写临时 SQL 查询对当前数据进行校验时，使用通配符是很合情合理的。只执行一次的查询对可维护性没有任何要求。

本书中的例子用到了通配符，一来是为了节省空间，二来是为了避免分散读者对例子中那些更重要部分的注意力。在实际的工作代码中，我很少使用通配符。

如果你的程序需要在增加、删除、重命名或者重新配置列时依旧能自动适应及调整，那最好还是使用通配符，但要确认对之前描述的那些陷阱有充分的准备。

你可以对一个联结查询中的每个独立的表使用通配符。在通配符之前加上表名或者别名作为前缀。这么做可以让你在从一张表中获取所有列的同时，从另一张表中获取少量你所指定的列。如下例：

Implicit-Columns/legit/wildcard-one-table.sql

```
SELECT b.*, a.first_name, a.email
FROM Bugs b JOIN Accounts a
  ON (b.reported_by = a.account_id);
```

输入一个很长的列的列表是很耗时的。对某些人来说，开发效率比程序执行效率更重要。类

似地，你可能会把“写更短更可读的查询语句”的优先级提高。使用通配符确实能减少输入的量，得到一个更简短的查询，因此，如果你确实注重这方面的需求，那就用通配符吧。

我听过一个开发人员抱怨说，从程序传递到 SQL 查询请求的包太大而使得网络负载加剧，在某些情况下查询语句的长度的确会造成影响。但更常见的情况是，返回的数据所使用的带宽比查询语句本身要多得多。你需要自己判断这些特殊的情况，别纠结于这些小问题。

19.5 解决方案：明确列出列名

每次查询时都列出所有你需要的列，而不是使用通配符或者隐式列的列表。

Implicit-Columns/soln/select-explicit.sql

```
SELECT bug_id, date_reported, summary, description, resolution,
  reported_by, assigned_to, verified_by, status, priority, hours
FROM Bugs;
```

Implicit-Columns/soln/insert-explicit.sql

```
INSERT INTO Accounts (account_name, first_name, last_name, email,
  password_hash, portrait_image, hourly_rate)
VALUES ('bkarwin', 'Bill', 'Karwin', 'bill@example.com',
  SHA2('xyzzy'), NULL, 49.95);
```

所有这些输入看上去都是很繁重的工作，但非常值得。

19.5.1 预防错误

还记得 poka-yoke[①]吗？你在查询时指明所需要选择的列，这能让 SQL 查询更好地应付错误以及更早地暴露问题。

- 如果这张表中某一列的位置被移动过，它不会对返回结果中这一列的位置造成影响。
- 如果这张表中新加入一列，它是不会出现在查询结果中的。
- 如果从这张表中删除一列，你的查询会得到一个错误——但是这样挺好，因为你直接就能定位到出错的查询语句，而不是在事后追查问题的起因。

如果指定了列名，在使用 `INSERT` 语句时也能得到类似的好处。你所定义的插入列的顺序会覆盖原始表的定义，并且插入的值会分配到你想要插入的列里。没有在列表里出现的新加入的那列，会自动获得一个默认值或者直接等于 `NULL`。如果你引用了一个已经被删除的列，就会得到一个错误信息，这能更早地发现并解决问题。

这是一个尽早出错原则的例子。

① 日本工业领域所使用的一种防差错技术，参考第 5 章。

19.5.2 你不需要它

如果必须关心软件的可扩展性和程序的吞吐量，你应该检查一下在网络传输过程中可能造成的浪费。在开发和测试环境中，SQL 查询所造成的流量上的问题可以忽略不计，但在生产环境中每秒上千次的 SQL 查询就会造成严重的问题。

一旦你禁止了 SQL 通配符，就很自然地有针对性地去除那些你不需要的列，同时也意味着更少的输入。这也能使得网络带宽的使用更加有效率。

Implicit-Columns/soln/yagni.sql

```
SELECT date_reported, summary, description, resolution, status, priority
FROM Bugs;
```

19.5.3 无论如何你都需要放弃使用通配符

你从自动售货机买了一包 M&M's，包装袋很有用，它能让你很简单地就把这些糖带回办公桌。一旦你打开了包装袋，就需要将 M&M's 的每一粒糖都视为独立的个体。它们会滚得到处都是。如果你不小心，一些糖还会掉在桌子下面引来臭虫。但是，你想吃到它们就只有打开包装袋。

在 SQL 查询中，一旦你想要对某一列进行一些表达式计算，或者使用一个列别名，或者由于性能原因排除某一列，你就打开了通配符这个"包装袋"。你不再享有将所有列的集合当成一个包处理所带来的便利，但能够访问到这个包里面的所有内容。

你不可避免地要在查询中引入列别名、函数，或者从列表中排除某列。如果你从一开始就不使用通配符，那之后要对查询进行修改就会变得更加方便。

随便拿，但是拿了就必须吃掉。

第20章 明文密码

敌人也了解这套系统。

▶ 香农[①]的格言

你接到一个电话，某个人在登录你提供技术支持的程序时遇到了麻烦。

“我是销售部的 Pat Johnson。我忘记了我的密码。你能帮我查一下密码是什么吗？”Pat 的声音听起来有点怪，还有点急和不好意思。

“对不起，我不能这么做。”你回答道：“我可以帮你重置密码，然后给你的注册邮箱发一封邮件，你可以根据 E-mail 里的指示重新设置密码。”

电话那头的男人变得更加不耐烦且不可理喻了。“真荒唐！”他说，“我上一家公司的技术支持就可以直接帮我看一下密码是什么。你难道连你自己的工作都做不好？你要我打电话给你的经理吗？”

你自然想要和你的用户保持良好的关系，因此你执行了一个 SQL 查询看了下 Pat Johnson 的账号密码，在电话上告诉了他。

这个人挂了电话后，你向同事说道：“真悬啊。我差点就收到 Pat Johnson 的投诉了。我希望他不会再抱怨了。”

你的同事很困惑。“他？销售部的 Pat Johnson 是个女的啊。我觉得你可能把她的密码给了一个骗子。”

20.1 目标：恢复或重置密码

我敢打赌，每个有密码的程序都会碰到用户忘记密码的情况。现今大多数程序都通过 E-mail

① Shannon，美国数学家，信息论的创始人。——编者注

的回馈机制让用户恢复或者重置密码。这个解决方案有一个前提，就是这个用户能访问他在注册服务时留下的邮箱。

20.2 反模式：使用明文存储密码

在这种恢复密码的解决方案中，很常见的一个错误是允许用户申请系统发送一封带有明文密码的邮件。这是数据库设计上一个可怕的漏洞，并且它会导致一系列安全问题，可能会使得未取得授权的人获得系统访问权限。

接下来的几段我们就来分析这些风险。假设我们的错误跟踪系统有一张 Accounts 表，每个用户的账号信息都是这张表里的一条记录。

20.2.1 存储密码

在 Accounts 表中，我们以最典型的字符串形式存储密码：

Passwords/anti/create-table.sql

```
CREATE TABLE Accounts (
  account_id    SERIAL PRIMARY KEY,
  account_name  VARCHAR(20) NOT NULL,
  email         VARCHAR(100) NOT NULL,
  password      VARCHAR(30) NOT NULL
);
```

你可以很简单地通过插入一条带有指定密码的记录来创建一个新账号：

Passwords/anti/insert-plaintext.sql

```
INSERT INTO Accounts (account_id, account_name, email, password)
  VALUES (123, 'billkarwin', 'bill@example.com', 'xyzzy');
```

使用明文存储密码或者使用明文在网络上传递密码都是不安全的。如果攻击者能够截获你用来插入密码的 SQL 语句，他们就能直接读到密码。在更新密码或者验证用户是否输入正确的密码时这么做，也会导致同样的问题。黑客有好几种方法能够盗取用户密码，比如下面几种。

- 在客户端和服务器端数据库交互的网络线路上截获数据包。这么做比你想象的要容易得多。有很多这样的软件能做到这一点，比如 Wireshark[①]。
- 在数据库服务器上搜索 SQL 的查询日志。要这么做的前提是，黑客能够访问到数据库所在的服务器，假设他们真的能登录到服务器上，他们就可以查看那些带有 SQL 语句的数据库执行日志。
- 从服务器或者备份介质上读取数据库备份文件内的数据。你的备份文件妥善保管了吗？你在回收或者丢弃备份设备之前彻底清理干净里面的数据了吗？

---

① Wireshark（也叫 Ethereal），其官网为 http://www.wireshark.org/。

20.2.2 验证密码

一段时间后，当用户想要登录，你的程序需要比较用户输入的密码和数据库中存储的密码是否一致。由于密码本身是以明文存储的，所以这样的比较也是以明文字符串的形式进行的。比如，你可以使用如下的查询来返回 0 或 1，判断用户输入的密码和数据库中的是否一致：

Passwords/anti/auth-plaintext.sql

```
SELECT CASE WHEN password = 'opensesame' THEN 1 ELSE 0 END
  AS password_matches
FROM Accounts
WHERE account_id = 123;
```

在上面的例子中，用户输入的密码 opensesame 是错的，因此整个查询返回了 0。

就像上一段存储密码中所说的那样，将用户输入的字符串以明文的形式插入到 SQL 语句中会让攻击者更容易得手。

别把两个不同的候选项绑在一起

大多数时候，我所看到的认证查询是将 account_id 和 password 两列同时放在 WHERE 子句里进行匹配查找：

Passwords/anti/auth-lumping.sql

```
SELECT * FROM Accounts
WHERE account_name = 'bill' AND password = 'opensesame';
```

在账号不存在或者用户输入的密码不正确时，整个查询返回空。你的程序无法区分到底认证为什么失败了。最好是使用一个能区分这两种情况的查询方法。那样就可以根据错误合适地选择处理方式了。

比如，当发现在短时间内同一个账号有很多失败的登录请求时，你可能会想暂时冻结这个账号，因为这可能是一次恶意攻击。然而，所面临的问题是你使用的查询语句没办法区分到底是用户名输错了，还是密码输错了。

20.2.3 在E-mail中发送密码

由于密码在数据库中是以明文形式存储的，你可以很简单地在程序中获取密码：

Passwords/anti/select-plaintext.sql

```
SELECT account_name, email, password
FROM Accounts
WHERE account_id = 123;
```

随后你的程序就可以根据用户的请求将密码发送到用户的邮箱里。你可能在访问过网站的密码提醒功能里看到过这样的邮件。比如下面这种类型：

密码恢复邮件实例:

```
From: daemon
To: bill@example.com
Subject: password request
```

你名为“bill”的账户请求密码提醒。

你的密码是"xyzzy"。

点击如下链接登录你的账户：

http://www.example.com/login

将明文密码通过邮件发送是非常严重的安全隐患。E-mail 可能会被黑客劫持、记录或者使用多种方式存储。就算使用安全协议查看邮件，收发邮件的服务由值得信赖的系统管理员维护，也不能保证一定安全。由于 E-mail 的收发都需要经由网络层传输，数据可能会在其他的路由节点上被截获。E-mail 安全协议的覆盖面并不足够广，也不是你能控制的。

20.3 如何识别反模式

任何能够恢复你的密码，或者将你的密码通过邮件以明文或可逆转加密的格式发给你的程序，都必然犯了本章的反模式。如果你的程序可以通过一个合法的方式获得用户的明文密码，那么黑客也可以做到同样的事情，只不过是不合法而已。

20.4 合理使用反模式

程序开发的道德标准

如果你在开发一个需要密码的程序，被要求设计一个恢复密码的特性，你应该拒绝这样的需求，提醒项目决策者这么做的风险，然后提供另一个能起到同样作用的解决方案。

正如一个电工应该能辨别并改正一个会导致火灾的接线设计一样，作为一个软件工程师，有责任和义务去了解相关的安全问题，并提升软件的安全性。

你可以阅读关于软件安全方面的 *19 Deadly Sins of Software Security*[HLV05]一书。另一个相关资源是 Open Web Application Security 项目（http://owasp.org）。

你的程序可能需要使用密码来访问一个第三方的服务——这意味着，你的程序可能是一个客户端，必须用可读的格式来存储这个密码。较好的做法是，使用一些程序能够解码的加密方法来存储，而不是直接使用明文的方式存储在数据库中。

你可以将身份认证和验证区分开来。用户可以随意说他自己是谁，但验证的逻辑就是用来证明他确实是他自己。密码就是最常被用来做验证这件事情的。

如果不能确保系统足够安全能抵御有技巧和有针对性的攻击，那么实际上系统只有认证机制而没有可靠的验证机制。但这并不一定是必须的。

并不是所有的程序都有被攻击的风险，也不是所有的程序都有敏感的需要保护的信息。比如说，一个可能只有几个可靠的内部人员访问的内部程序，认证机制就可能足够了。在那些非正式的环境中，一个简单的登录框就已经足够了。额外的建立一个强验证系统可能并不合理。

尽管如此，还是要小心——程序的使用总是有超出它们原来设定环境和作用的趋势。在你的小型内部程序暴露在公司防火墙之外之前，你应该要找个安全专家来评估一下它的安全性。

20.5 解决方案：先哈希，后存储

本章的反模式所描述的主要问题是密码的原始存储格式是可读的。其实可以不将密码读出来就验证用户输入的密码是否正确。这一节就介绍了怎样在 SQL 数据库中实现这样的安全存储密码的方案。

20.5.1 理解哈希函数

使用单向哈希函数对原始密码进行加密，哈希是指将输入字符串转化成另一个新的、不可识别的字符串的函数。使用哈希函数后，连原始输入串的长度也变得难以猜测了，因为哈希函数返回的字符串的长度是固定的。比如，SHA-256 算法将我们的密码 xyzzy，转换成了一个 256 位的串，若使用 16 进制表示是一个 64 字节的字符串。

```
SHA2('xyzzy') = '184858a00fd7971f810848266ebcecee5e8b69972c5ffaed622f5ee078671aed'
```

哈希的另一个特征就是不可逆。由于哈希函数的算法设计就是要“丢失”一些输入串的信息，所以你无法从一个哈希串恢复出原始输入串。一个好的哈希算法应该需要花上和直接猜测密码差不多的工作量才能找到原始串。

曾经比较流行的哈希算法是 SHA-1，但研究者最近证明，这个只有 160 位的哈希算法还不够安全，有一种技术能通过哈希串推算出输入串。这项技术非常耗时，但无论如何，都比靠猜破解密码所花的时间短。美国国家标准和技术协会（NIST）宣布，2010 年后开始逐步取消 SHA-1 作为安全哈希算法的资格，取而代之的是其更强大的变异算法[①]：SHA-224、SHA-256、SHA-384 和 SHA-512。无论是否遵循 NIST 的标准，至少使用 SHA-256 算法加密密码总是好的。

① http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html。

MD5 是另一个流行的哈希函数，产生 128 位的哈希串。MD5 也被证明是弱加密，因此你最好不要用它来加密密码。稍弱的算法还是有很多使用场景的，但对于诸如密码一类的敏感信息，它们并不适用。

20.5.2 在SQL中使用哈希

下面是对 Accounts 表的重定义。SHA-256 的哈希密码总是一个 64 字节的字符串，因此这一列的类型是固定长度的 CHAR。

Passwords/soln/create-table.sql

```
CREATE TABLE Accounts (
  account_id     SERIAL PRIMARY KEY,
  account_name   VARCHAR(20),
  email          VARCHAR(100) NOT NULL,
  password_hash  CHAR(64) NOT NULL
);
```

哈希函数并不是标准的 SQL 语言，因此你可能要依赖于所使用数据库提供的哈希扩展。比如，MySQL 6.0.5 的 SSL 扩展支持包含了 SHA2()的函数，它默认返回一个 256 位的哈希串。

Passwords/soln/insert-hash.sql

```
INSERT INTO Accounts (account_id, account_name, email, password_hash)
  VALUES (123, 'billkarwin', 'bill@example.com', SHA2('xyzzy'));
```

你可以在存储和验证用户输入时使用同样的哈希算法，并对两个哈希后的值进行比较。

Passwords/soln/auth-hash.sql

```
SELECT CASE WHEN password_hash = SHA2('xyzzy') THEN 1 ELSE 0 END
  AS password_matches
FROM Accounts
WHERE account_id = 123;
```

你可以通过简单地将用户的密码改成一个永远不可能通过哈希函数得到的字符串将其锁住。比如，noaccess 这个字符串包含了非十六进制字符，是不可能由哈希函数返回的。

20.5.3 给哈希加料

如果你使用哈希串替代了原始密码，然后攻击者获得了对数据库的访问权限（比如他翻了你的垃圾桶，找到了被丢弃的备份 CD），他仍旧可以通过试错法获取用户密码。要猜出每个密码可能会花很长时间，但他可以预先准备好自己的数据库——存储可能的密码和对应的哈希串，然后和从你的数据库中找到的哈希串进行比较。只要有一个用户选择了字典中存在的单词，攻击者就能够很轻易地通过搜索两边的哈希值来找到对应的密码原文。他甚至可以直接使用 SQL 来做这件事：

Passwords/soln/dictionary-attack.sql

```
CREATE TABLE DictionaryHashes (
  password       VARCHAR(100),
  password_hash  CHAR(64)
);

SELECT a.account_name, h.password
FROM Accounts AS a JOIN DictionaryHashes AS h
  ON a.password_hash = h.password_hash;
```

防御这种“字典攻击”的一种方法是给你的密码加密表达式加点佐料。具体方法是在将用户密码传入哈希函数进行加密之前，将其和一个无意义的串拼接在一起，即使用户选择了一个在字典中存在的单词作为密码，对加料密码进行哈希得到的串是不太会出现在攻击者的哈希数据库中的，你可以发现增加了随机串得到的哈希值和原始值是不一样的：

```
SHA2('password')
  = '5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8'

SHA2('password-0xT!sp9')
  = '7256d8d7741f740ee83ba7a9b30e7ac11fcd9dbd7a0147f4cc83c62dd6e0c45b'
```

每个密码都应该配上不同的随机串，这样攻击者就必须为每个密码都创建一个新的哈希字典。然后他就会回到起点上，因为破解数据库中的密码所花的时间和靠猜达到目的的时间差不多[①]。

Passwords/soln/salt.sql

```
CREATE TABLE Accounts (
  account_id     SERIAL PRIMARY KEY,
  account_name   VARCHAR(20),
  email          VARCHAR(100) NOT NULL,
  password_hash  CHAR(32) NOT NULL,
  salt           BINARY(8) NOT NULL
);

INSERT INTO Accounts (account_id, account_name, email,
    password_hash, salt)
  VALUES (123, 'billkarwin', 'bill@example.com',
    SHA2('xyzzy' || '-0xT!sp9'), '-0xT!sp9');
SELECT (password_hash = SHA2('xyzzy' || salt)) AS password_matches
FROM Accounts
WHERE account_id = 123;
```

佐料的合理长度应该是 8 个字节。你需要为每个密码随机生成佐料。之前的几个例子里，佐料字符串使用的都是可打印字符，但事实上你可以使用随机的、不可打印的任意字符。

① 从哈希值恢复密码的一种更优雅的技术叫做彩虹表，它的性能令人吃惊，但引入随机串也能防御这种技术。

20.5.4 在SQL中隐藏密码

现在，你在存储密码之前已经有了一个强哈希函数来对密码进行加密，并且使用了随机字符串来阻止字典攻击，你可能认为这样已经足够安全了。但是，密码还是会在 SQL 表达式中以明文的形式出现，这意味着如果攻击者截获了网络通信的数据包，或者记录了相关的查询语句的日志给到了错误的人手里，密码就泄露了。

只要不将明文密码放到 SQL 查询语句中，就能避免这种类型的泄露。你所需要做的是在程序代码中生成密码的哈希串，然后在 SQL 查询中使用哈希串。这么做的好处是，即使攻击者截获了数据包，他也没办法将哈希反转成他所需要的密码。

即使这么做，你还是需要在哈希之前给原始密码追加随机字符串。

下面是一个 PHP 的例子，使用了 PDO 扩展来获得佐料、计算得到哈希串，然后执行查询请求来和数据库中存储的加料哈希值进行比较：

Passwords/soln/auth-salt.php

```
<?php

$password = 'xyzzy';

$stmt = $pdo->query(
   "SELECT salt
    FROM Accounts
    WHERE account_name = 'bill'");

$row = $stmt->fetch();
$salt = $row[0];

$hash = hash('sha256', $password . $salt);

$stmt = $pdo->query("
  SELECT (password_hash = '$hash') AS password_matches;
  FROM Accounts AS a
  WHERE a.acct_name = 'bill'");
$row = $stmt->fetch();
if ($row === false) {
  // account 'bill' does not exist
} else {
  $password_matches = $row[0];
  if (!$password_matches) {
    // password given was incorrect
  }
}
```

这个 hash()函数能确保返回的一定是十六进制数，因此不会有 SQL 注入的风险（详情参阅第 21 章）。

在网络程序中，还有另一个地方是攻击者有机会截获网络数据包的：在用户的浏览器和网站服务器之间。当用户提交了一个登录表单时，浏览器将用户的密码以明文形式发送到服务器端，随后服务器端才能使用这个密码进行之前所介绍的哈希运算。你可以通过在用户的浏览器发送表单数据之前就进行哈希运算来解决这个问题。但这个方案也有一些不足的地方，就是你需要在进行正确的哈希运算之前，还要通过别的途径获得和这个密码相关联的佐料。折中方案就是在从浏览器向服务器端提交密码表单时，使用安全的 HTTP（https）链接。

20.5.5 重置密码，而非恢复密码

现在，密码已经以一个更安全的方法存储了，但你还是需要解决最原始的问题：帮助那些忘记密码的用户。由于现在数据库中存着的密码是哈希串而不是原始密码，你已经无法恢复他们的密码了。你没办法比一个攻击者更快速地反转一个哈希值，但可以允许用户用别的途径获得访问权限。这里介绍两个简单方案。

第一个方案是当用户忘记他们的密码请求帮助的时候，程序发送一封带有临时生成密码的邮件给用户，而不是直接发给他自己的密码。为了安全起见，在一个较短的时间之内，这个临时密码就会过期，即使这封 E-mail 被截获了，程序还是不会允许未验证的访问。同时，程序应该设计成一旦用户使用临时密码登录后，就应该被强制要求修改密码。

系统生成临时密码邮件

```
From: daemon
To: bill@example.com
Subject: password reset

你要求重置你账户的密码。

你的临时密码是"p0trz3ble"。

一小时之后，这个密码将不能使用。

点击如下链接登录你的账号并设置你的密码：

http://www.example.com/login
```

第二个方案，在数据库中记录下这个请求，并且为其分配一个唯一的令牌作为标识，而不是发送带有新密码的邮件：

Passwords/soln/reset-request.sql

```
CREATE TABLE PasswordResetRequest (
  token       CHAR(32) PRIMARY KEY,
  account_id  BIGINT UNSIGNED NOT NULL,
```

```
  expiration  TIMESTAMP NOT NULL,
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

SET @token = MD5('billkarwin' || CURRENT_TIMESTAMP);

INSERT INTO PasswordResetRequest (token, account_id, expiration)
  VALUES (@token, 123, CURRENT_TIMESTAMP + INTERVAL 1 HOUR);
```

随后你可以在 E-mail 中包含这个令牌。你也可以使用别的途径发送这个令牌，比如短信，只要能将消息送达到这个请求重置密码的账号所有者即可。使用这种方法，如果一个陌生人非法地请求了一次密码重置，系统只会发送 E-mail 给这个账号的实际拥有者。

密码重置页面临时链接邮件

```
From: daemon
To: bill@example.com
Subject: password reset
你要求重置你账户的密码。
在一小时内点击下面链接来改变密码。
一小时之后，这个链接将无法工作，你的密码保持不变。
http://www.example.com/reset_password?token=f5cabff22532bd0025118905bdea50da
```

当程序收到一个从重置密码页面来的请求，令牌的值必须存在于密码重置请求表中，并且该行的过期时间点必须是一个将来的时间点而不是过去的时间点。同时，该行的 account_id 引用 Accounts 表，因此，这个令牌被约束为只能重置一个指定的账号。

当然，如果其他人访问了这个页面，也会造成问题。可以通过一些简单的方法来减小风险，比如这个特殊页面的有效期非常短，并且这个页面上不会显示哪个账号的密码要被重置等。

密码学在不断进步，努力保持领先于攻击技术。本章所介绍的这些技术能够帮助改进很多一般的程序，但如果你所开发的系统对安全性要求非常高，你应该深入研究一些更高级的技术，比如下面提到的。

- PBKDF2（http://tools.ietf.org/html/rfc2898）是一个被广泛使用的密码加强标准。
- Bcrypt（http://bcrypt.sourceforge.net/）实现了一个自适应哈希函数。

如果密码对你可读，那对攻击者也如此。

第21章 SQL注入

就像我说的，我被错误地引述了。

► 格劳乔·马克思

2010 年 3 月，美国历史上最大规模的身份盗窃案告破，黑客 Albert Gonzalez 伏法。他通过侵入 ATM 机、几家大型零售商的系统以及相关联的信用卡公司的系统，总计盗窃了 1.3 亿个信用卡与借记卡信息。

Gonzalez 打破了他自己保持的盗窃记录，原来的记录是他使用了无线网络的漏洞，在 2006 年盗窃了 4560 万个信用卡与借记卡信息。

Gonzalez 是怎么能做到的？我们可以想象一下邦德电影里的场景，从电梯井放下绳索，使用超级计算机，破解最先进的加密算法，或者破坏整座城市的电力系统之类的。

起诉书中对案情的描写要真实平常得多。Gonzalez 利用了互联网上最常见的安全漏洞。他使用了 SQL 注入的攻击手段，获得了往受害服务器上传文件的权限，随后他和同伙获得了对系统的访问权限。起诉书[①]中有如下这样的描述。

攻击执行手段：恶意软件

> 他们会在受害者的电脑上安装一个叫做 sniffer 的程序，这个程序会在受害人使用网络进行交易时，收集他的信用卡账号和相关信息，然后定期地将这些数据发送给这伙人。

被 Gonzalez 攻击的那几家零售商声称，他们已经修复了这些安全漏洞。然而，他们堵住了一个漏洞，每天都有新的带有其他漏洞的网络程序被发布。SQL 注入对于黑客来说仍然是一个很容易的突破口，因为软件开发人员并不了解这个漏洞的原理，以及该如何防止这样的攻击。

① http://voices.washingtonpost.com/securityfix/heartlandIndictment.pdf。

21.1 目标：编写 SQL 动态查询

SQL 常和程序代码一起使用。我们通常所说的 SQL 动态查询[①]，是指将程序中的变量和基本 SQL 语句拼接成一个完整的查询语句。

SQL-Injection/obj/dynamic-sql.php

```
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id = $bug_id";
$stmt = $pdo->query($sql);
```

这个简单的例子展示了如何将 PHP 变量插入到一个 SQL 查询语句中。我们期望`$bug_id`是一个整型，因此当数据库接收到这个请求时，`$bug_id`的值就是查询语句的一部分。

SQL 动态查询是有效利用数据库的很自然的方法。当你使用程序内的变量来指定如何进行查询时，就是在将 SQL 作为链接程序和数据库的桥梁。程序和数据库之间通过这种方式进行“对话”。

然而，要让程序按照你想要的方式执行并不难，难的是要让程序变得安全，不执行你不想让它执行的操作。但软件在受到 SQL 注入攻击时，通常都无法保证安全。

21.2 反模式：将未经验证的输入作为代码执行

当往 SQL 查询的字符串中插入别的内容，而这些被插入的内容以你不希望的方式修改了查询语句的语法时，SQL 注入就成功了。在传统的 SQL 注入案例中，所插入的内容首先完成一个查询，然后再执行第二个完整的查询逻辑。比如，如果`$bug_id`的值是`1234; DELETE FROM Bugs`，之前例子中的查询语句最终会变成这样：

SQL-Injection/anti/delete.sql

```
SELECT * FROM Bugs WHERE bug_id = 1234; DELETE FROM Bugs
```

这种类型的 SQL 注入比较明显，就如图 21-1[②]的这个故事一样。通常这些缺陷隐藏得比较好，但还是会有危险。

---

① 技术上来说，任何在运行时解析的查询都是 SQL 动态查询，但通常在实际中，SQL 动态查询指的是在语句中包含变量数据的查询请求。

② 漫画由 Randall Munroe 所绘（http://xkcd.com/327）已经许可。

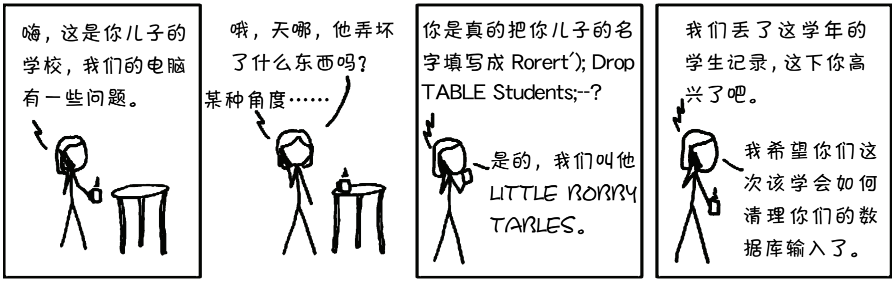


图 21-1　妈妈的功绩

21.2.1　意外无处不在

假设你正在为 Bug 数据库编写一个 Web 程序，其中的一个页面允许你根据名字访问项目：

SQL-Injection/anti/ohare.php

```
<?php
$project_name = $_REQUEST["name"];
$sql = "SELECT * FROM Projects WHERE project_name = '$project_name'";
```

当你的开发小组开始为芝加哥的 O'Hare 国际机场开发软件时，问题出现了。你们很自然地给这个项目取名为“O'Hare”，在你们的 Web 程序中，要怎样提交一个请求才能看到这个项目呢？

```
http://bugs.example.com/project/view.php?name=O'Hare
```

你的 PHP 代码接受了请求参数并插入到 SQL 查询语句中，但是实际生成的 SQL 语句却不是大家所期望的：

SQL-Injection/anti/ohare.sql

```
SELECT * FROM Projects WHERE project_name = 'O'Hare'
```

由于字符串是被两个单引号所包围，生成的 SQL 语句中包含了一个很短的`'O'`，然后是一些额外的字符，在这个例子中，`Hare'`没有任何意义。对于这样的 SQL 语句，数据库只能返回一个语法错误的提示。这的确是一个意外。发生任何不好的事情的风险还是比较小的，因为语法错误的 SQL 语句是不会被执行的。风险较大的情况是产生的 SQL 没有任何语法错误，并且以一种你所不希望的方式执行。

21.2.2　对Web安全的严重威胁

当攻击者能够使用 SQL 注入操控你的 SQL 查询语句时，它就变成了一个巨大的威胁。比如，你的程序可能允许用户以如下方式修改密码：

SQL-Injection/anti/set-password.php

```
<?php
$password = $_REQUEST["password"];
$userid = $_REQUEST["userid"];
$sql = "UPDATE Accounts SET password_hash = SHA2('$password')
    WHERE account_id = $userid";
```

一个聪明的攻击者会猜测你的每个请求参数对应在 SQL 语句中的作用，并且精心选择每个参数对应的值：

```
http://bugs.example.com/setpass?password=xyzzy&userid=123 OR TRUE
```

通过在 `userid` 这个参数后插入额外的字符串，攻击者改变了对应的 SQL 语句的意义。现在，这条语句会改变每一个用户的密码，而不是只改变一个用户：

SQL-Injection/anti/set-password.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = 123 OR TRUE;
```

这是理解 SQL 注入的关键，也是如何防止 SQL 注入的关键：SQL 注入是通过在 SQL 语句被数据库解析之前，以修改其语法的形式工作的。只要你在解析语句之前插入动态部分，就存在 SQL 注入的风险。

有数不尽的方法选择一个恶意的字符串来改变 SQL 语句的行为。它只受制于攻击者的想象力和你保护 SQL 语句的能力。

21.2.3 寻找治愈良方

既然我们已经知道了 SQL 注入的风险，下一个问题便是：要做什么来使代码远离 SQL 注入呢？可能你之前读过一些博文或者一些文章，声称某一种技术是对抗 SQL 注入的万能药。而事实上，这些技术都被证明无法阻挡所有类型的 SQL 注入。因此你需要在不同的情况下将所有这些技术组合起来使用。

转义

防止 SQL 语句包含任何不匹配的引号的最古老的方法，就是对所有的引号字符进行转义操作，使它们不至于成为字符串的结束符。在标准 SQL 中，可以使用两个连续的单引号来表示一个单引号字符：

SQL-Injection/anti/ohare-escape.sql

```
SELECT * FROM Projects WHERE project_name = 'O''Hare'
```

大多数的数据库还支持使用反斜杠对单引号字符进行转义操作，就和大多数其他的编程语言一样：

SQL-Injection/anti/ohare-escape.sql

```
SELECT * FROM Projects WHERE project_name = 'O\'Hare'
```

这么做的原理是，在将应用程序中的数据插入到 SQL 语句之前就进行转换。大多数 SQL 的编程接口都会提供一个简便的函数来做这个操作。比如，在 PHP 的 PDO 扩展中，可以使用一个 quote()函数来定义一个包含引号的字符串或者还原一个字符串中的引号字符。

SQL-Injection/anti/ohare-escape.php

```
<?php
$project_name = $pdo->quote($_REQUEST["name"]);
$sql = "SELECT * FROM Projects WHERE project_name = $project_name";
```

这种技术能够减少由于动态内容中不匹配的引号所造成的 SQL 注入的风险。但在非字符串内容的情况下，这种技术就会失效。

SQL-Injection/anti/set-password-escape.php

```
<?php
$password = $pdo->quote($_REQUEST["password"]);
$userid = $pdo->quote($_REQUEST["userid"]);
$sql = "UPDATE Accounts SET password_hash = SHA2($password)
    WHERE account_id = $userid";
```

SQL-Injection/anti/set-password-escape.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = '123 OR TRUE'
```

你没办法让一个数值列直接和一个带有数字的字符串进行比较，不管使用哪款数据库，这都是不可能的。某些数据库可能会隐式地将字符串转换成一个等价的数字，但在标准 SQL 中，在将字符串转换成数字时，一定要明确使用 CAST()函数。

在一些案例中，有些使用非 ASCII 编码的字符串在经过了对引号字符进行转义的函数的处理后，依旧保持引号字符没有任何改变。[①]

查询参数

一个经常被认为是防止 SQL 注入的万能型解决方案是使用*查询参数*。不同于在 SQL 语句中插入动态内容，查询参数的做法是在准备查询语句的时候，在对应参数的地方使用*参数占位符*。随后，在执行这个预先准备好的查询时提供一个参数。

SQL-Injection/anti/parameter.php

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Projects WHERE project_name = ?");
$params = array($_REQUEST["name"]);
$stmt->execute($params);
```

① 可以参考 http://bugs.mysql.com/8378。

大多数开发人员都推荐这个方案，因为你不需要对动态内容进行转义，或者担心有缺陷的转义函数。事实上，查询参数这个方法的确是应对 SQL 注入的强劲解决方案。但是这还不是一个通用的解决方案，因为查询参数总被视为是一个字面值。

- 多个值的列表不可以当成单一参数：

  SQL-Injection/anti/parameter.php

  ```
  <?php
  $stmt = $pdo->prepare("SELECT * FROM Bugs WHERE bug_id IN ( ? )");
  $stmt->execute(array("1234,3456,5678"));
  ```

  这样的做法会导致数据库认为传入的是一个包含数字和逗号的字符串，处理过程和将一系列整数作为参数进行查询并不一样：

  SQL-Injection/anti/parameter.sql

  ```
  SELECT * FROM Bugs WHERE bug_id IN ( '1234,3456,5678' )
  ```

- 表名无法作为参数：

  SQL-Injection/anti/parameter.php

  ```
  <?php
  $stmt = $pdo->prepare("SELECT * FROM ? WHERE bug_id = 1234");
  $stmt->execute(array("Bugs"));
  ```

  这么做是想将一个字符串插入表名所在的位置，但只会得到一个语法错误的提示：

  SQL-Injection/anti/parameter.sql

  ```
  SELECT * FROM 'Bugs' WHERE bug_id = 1234
  ```

- 列名无法作为参数：

  SQL-Injection/anti/parameter.php

  ```
  <?php
  $stmt = $pdo->prepare("SELECT * FROM Bugs ORDER BY ?");
  $stmt->execute(array("date_reported"));
  ```

  在这个例子中，排序是一个无效操作，因为排序表达式是一个常量字符串，对所有行都是一样的：

  SQL-Injection/anti/parameter.sql

  ```
  SELECT * FROM Bugs ORDER BY 'date_reported';
  ```

- SQL 关键字不能作为参数：

  SQL-Injection/anti/parameter.php

  ```
  <?php
  ```

```
$stmt = $pdo->prepare("SELECT * FROM Bugs ORDER BY date_reported ?");
$stmt->execute(array("DESC"));
```

参数将被当做字面字符串插入而非 SQL 关键字。在这个例子中，会返回语法错误的提示。

SQL-Injection/anti/parameter.sql

```
SELECT * FROM Bugs ORDER BY date_reported 'DESC'
```

我的查询是如何完成的

很多人认为 SQL 的查询参数就是一种能自动将参数包含进 SQL 语句的技术。这并不准确，如果仅仅这么想，会导致对这项技术的严重误解。

RDBMS 服务器首先会解析你所准备好的 SQL 查询语句。在完成这一步操作之后，就没有任何方法能够改变那句 SQL 查询语句的结构。

在执行一个已经准备好的 SQL 查询时，你需要提供对应的参数，每个你提供的参数都对应于预先准备好的查询中的一个占位符。

你可以重复执行这个预先准备好的查询，只要使用新的参数替换掉老的参数就行了。因此，RDBMS 系统必须将查询操作和对应的传入参数分开跟踪。这对于系统安全来说是一件好事。

这意味着，如果你获取到一个预先准备好的 SQL 查询语句，它里面是不会包含任何实际的参数值的。当你调试或者记录查询时，很方便就能看到带有参数值的 SQL 语句，但这些值永远不会以可读的 SQL 形式整合到查询中去。

调试动态化 SQL 语句的最好方法，就是将准备阶段的带有占位符的查询语句和执行阶段传入的参数都记录下来。

存储过程

存储过程，是很多程序员声称可以抵御 SQL 注入攻击的方法。通常来说，存储过程包含固定的 SQL 语句，这些语句是在定义这个存储过程的时候被解析的。

然而，在存储过程中也是可以使用 SQL 动态查询的，而这么做也会有安全隐患。在下面的这个例子中，`input_userid` 参数会被直接替换到 SQL 查询中，而这么做很不安全。

SQL-Injection/anti/procedure.sql

```
CREATE PROCEDURE UpdatePassword(input_password VARCHAR(20),
  input_userid VARCHAR(20))
BEGIN
  SET @sql = CONCAT('UPDATE Accounts
    SET password_hash = SHA2(', QUOTE(input_password), ')
    WHERE account_id = ', input_userid);
```

```
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
END
```

在存储过程中使用 SQL 动态查询，一点也不比在程序代码中直接使用 SQL 动态查询安全。这个 `input_userid` 参数可以包含有害的内容，并且造成最终执行的 SQL 语句变得不安全。

SQL-Injection/anti/set-password.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = 123 OR TRUE;
```

数据访问框架

你可能看过数据访问框架的拥护者声称他们的库能够抵御所有 SQL 注入的攻击。对于所有允许你使用字符串方式传入 SQL 语句的框架来说，这都是扯淡。

良好的规范

在我做了一个关于我所开发的基于 PHP 的数据访问框架的讲座之后，一位听众站起来问我："你的框架能抵挡 SQL 注入吗？"我告诉他，这个框架提供了一些处理字符串的函数和使用查询参数的方法。

这个年轻人看上去很疑惑。"但是，它能抵挡 SQL 注入吗?"他重复道。他在寻求一种自动化的方法来确保他没有犯任何自己都无从辨认的错误。

我告诉他使用框架来抵挡 SQL 注入，就好比使用牙刷来防止蛀牙。你需要持续使用才能有所帮助。

没有任何框架能强制你写出安全的 SQL 代码。一个框架可能会提供一系列简单的函数来帮助你，但很容易就能绕开这些函数，然后使用通常的修改字符串的办法来编写不安全的 SQL 语句。

21.3 如何识别反模式

事实上几乎所有的数据库应用程序都动态地构建 SQL 语句。如果你使用拼接字符串的形式或者将变量插入到字符串中的方法来构建哪怕一句 SQL 语句，那这一句查询语句就会让应用程序暴露在 SQL 注入攻击的威胁下。

SQL 注入攻击非常常见，因而你应该假设在程序中使用了 SQL 的部分总是存在那么一两个漏洞的。除非你的团队专门针对 SQL 注入做过一次代码审查，找到并且修复了相关的漏洞。

规则 31：检查车后座

如果你喜欢看怪物电影，就会知道那些可怕的生物总是喜欢藏在车后座，等着你坐进车里。这教导我们，千万别假设你所熟悉的地方就一定安全，比如你的车。

SQL 注入可以采用间接的形式。即使你最开始使用查询参数安全地插入用户提供的数据，你也可能在后续的查询中动态地使用这些数据：

```
<?php
$sql1 = "SELECT last_name FROM Accounts WHERE account_id = 123";
$row = $pdo->query($sql1)->fetch();
$sql2 = "SELECT * FROM Bugs WHERE MATCH(description) AGAINST ('"
  . $row["last_name"] .  "')";
```

在上一个查询中，如果用户使用了“O’Hara”这个用户名，或者故意包含了 SQL 语法的词，又会发生什么呢？

21.4　合理使用反模式

这个反模式和本书中其他的反模式不同，因为没有任何合理的理由允许 SQL 注入让程序存在安全漏洞。编写能够防御 SQL 注入的代码，以及帮助你的同事也这么做，是你作为一个软件开发人员的责任。软件的安全性是用其最薄弱的环节来衡量的——确保这个最薄弱的环节不是由你造成的！

21.5　解决方案：不信任任何人

没有哪一种技术能使 SQL 代码变得安全。你应该学习下面所描述的所有技术，并在合理的地方使用它们。

21.5.1　过滤输入内容

你应该将所有不合法的字符从用户输入中剔除掉，而不是纠结于是否有些输入包含了有危险的内容。也就是说，如果你需要一个整数，那就只使用输入中的整数部分。根据你所使用的开发语言不同，方法也不尽相同，比如在 PHP 中，可以使用 `filter` 扩展：

SQL-Injection/soln/filter.php

```
<?php
$bugid = filter_input(INPUT_GET, "bugid", FILTER_SANITIZE_NUMBER_INT);
$sql = "SELECT * FROM Bugs WHERE bug_id = {$bugid}";
$stmt = $pdo->query($sql);
```

对于简单的数字转换，你可以直接使用类型转换函数：

SQL-Injection/soln/casting.php

```
<?php
$bugid = intval($_GET["bugid"]);
$sql = "SELECT * FROM Bugs WHERE bug_id = {$bugid}";
$stmt = $pdo->query($sql);
```

你还可以使用正则表达式来匹配安全的子串，过滤非法内容：

SQL-Injection/soln/regexp.php

```
<?php
$sortorder = "date_reported"; // default

if (preg_match("/[_[:alnum:]]+/", $_GET["order"], $matches)) {
  $sortorder = $matches[1];
}

$sql = "SELECT * FROM Bugs ORDER BY {$sortorder}";
$stmt = $pdo->query($sql);
```

21.5.2 参数化动态内容

如果查询中的变化部分是一些简单的类型，你应该使用查询参数将其和 SQL 表达式分离。

SQL-Injection/soln/parameter.php

```
<?php
$sql = "UPDATE Accounts SET password_hash = SHA2(?) WHERE account_id = ?";
$stmt = $pdo->prepare($sql);
$params = array($_REQUEST["password"], $_REQUEST["userid"]);
$stmt->execute($params);
```

我们看到 21.2 节中，一个参数只能被替换为一个值。如果你是在 RDMBS 解析完 SQL 语句之后才插入这个参数值，没有哪种 SQL 注入的攻击能够改变一个参数化了的查询的语法结构。即使攻击者尝试使用带有恶意的参数值，诸如 `123 OR TRUE`，RDBMS 会将这个字符串当成一个完整的值插入。最坏情况下，这个查询没办法返回任何记录，它不会返回错误的行。这个恶意串最终可能会导致程序生成如下的查询语句，但这条语句无害：

SQL-Injection/soln/parameter.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = '123 OR TRUE'
```

当需要将程序中的变量以字符串形式插入 SQL 表达式中时，应该使用查询参数。

21.5.3 给动态输入的值加引号

查询参数通常来说是最好的解决方案，但在有些很特殊的情况下，参数的占位符会导致查询

优化器无法正确选择使用哪个索引来进行优化。

比如说，假设在 Accounts 表中有一个 is_active 列。这一列中 99%的记录都是真实值。对 is_active = false 的查询会得益于这一列上的索引，但对于 is_active = true 的查询却会在读取索引的过程中浪费很多时间。然而，如果你用了一个参数 is_active = ? 来构造这个表达式，优化器不知道在预处理这条语句的时候你最终会传入哪个值，因此很有可能就选择了错误的优化方案。

要规避这样的问题，直接将变量内容插入到 SQL 语句中会是更好的方法，不要去理会查询参数。一旦你决定这么做了，就一定要小心地引用字符串。

SQL-Injection/soln/interpolate.php

```
<?php
$quoted_active = $pdo->quote($_REQUEST["active"]);
$sql = "SELECT * FROM Accounts WHERE is_active = {$quoted_active}";
$stmt = $pdo->query($sql);
```

你需要确信所使用的函数是经过严格测试的、不会带有 SQL 安全隐患的。大多数的数据库访问库都包含一个这样的字符串引用处理函数。比如在 PHP 中，可以使用 PDO::quote()。别总想着自己实现一个这样的函数，除非你对所有的安全风险有深入的了解。

21.5.4 将用户与代码隔离

查询参数和转义字符能帮助你将字符串类型的值插入到 SQL 表达式中，但这些技术在需要插入表/列名或者 SQL 关键字的时候不起作用。你需要另一项技术来使得这些部分也能动态化。

假设你的用户想要能够选择以什么方式给 Bug 排序，比如按照状态或者按照提交的日期以及排序的方向。

SQL-Injection/soln/orderby.sql

```
SELECT * FROM Bugs ORDER BY status ASC
SELECT * FROM Bugs ORDER BY date_reported DESC
```

在如下的例子中，PHP 脚本接收 order 和 dir 两个参数，然后代码将用户的选择作为列名和关键字，插入到 SQL 查询中。

SQL-Injection/soln/mapping.php

```
<?php
$sortorder = $_REQUEST["order"];
$direction = $_REQUEST["dir"];
$sql = "SELECT * FROM Bugs ORDER BY $sortorder $direction";
$stmt = $pdo->query($sql);
```

这个脚本假设 order 字段包含了一个列名，dir 字段为 ASC 或者 DESC。但这并不是一个安

全的假设，因为用户在网络请求中可以随意传递参数的值。

参数化 IN()操作

我们知道你不能将一个由逗号分割的字符串当做单个参数传入，你需要跟你的对象列表一样多的参数。

比如需要通过主键来查询 6 个 Bug，你将这个列表存在数组变量`$bug_list`中：

```
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id IN (?, ?, ?, ?, ?, ?)");
$stmt = $pdo->prepare($sql);
$stmt->execute($bug_list);
```

仅当`$bug_list`中正好包含 6 个值，和占位符的数量完全匹配，上面这条语句才能正常工作。你应该动态地创建 IN()这个语句，使用和`$bug_list`中数据个数一样多的占位符。

下面的 PHP 例子使用了一些 PHP 内置的数组函数来生成一个占位符数组，其条目个数和`$bug_list`的条目个数一致，然后在插入到 SQL 表达式之前，将这些数组用逗号拼接在一起。

```
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id IN ("
    . join(",", array_fill(0, count($bug_list), "?")) . ")";
$stmt = $pdo->prepare($sql);
$stmt->execute($bug_list);
```

用这个技巧来参数化一个列表。

对应的解决方案是，将请求的参数作为索引值去查找预先定义好的值，然后用这些预先定义好的值来组织 SQL 查询语句。

(1) 声明一个`$sortorders`数组，将用户的可选项作为索引值，将 SQL 的列名作为对应的值。声明一个`$directions`数组，将用户的可选项作为索引，将 SQL 关键字 ASC 和 DESC 作为对应的值。

SQL-Injection/soln/mapping.php

```
$sortorders = array( "status" => "status", "date" => "date_reported" );
$directions = array( "up" => "ASC", "down" => "DESC" );
```

(2) 当用户的选择不在这两个数组中时，让变量`$sortorder`和`$dir`等于默认值。

SQL-Injection/soln/mapping.php

```
$sortorder = "bug_id";
$direction = "ASC";
```

(3) 如果用户的选择在$sortorders 和$directions 数组中，就使用对应的值。

SQL-Injection/soln/mapping.php

```
if (array_key_exists($_REQUEST["order"], $sortorders)) {
  $sortorder = $sortorders[ $_REQUEST["order"] ];
}

if (array_key_exists($_REQUEST["dir"], $directions)) {
  $direction = $directions[ $_REQUEST["dir"] ];
}
```

(4) 现在使用$sortorder 和$direction 这两个变量就是安全的了，因为它们只能是在代码中预先定义的值。

SQL-Injection/soln/mapping.php

```
$sql = "SELECT * FROM Bugs ORDER BY {$sortorder} {$direction}";
$stmt = $pdo->query($sql);
```

这种技术有如下几个优势。

- 从不将用户的输入与 SQL 查询语句连接，因此减少了 SQL 注入的风险。
- 可以让 SQL 语句中的任意部分变得动态化，包括标识、SQL 关键字，甚至整句表达式。
- 使用这个方法验证用户的输入变得很简单且高效。
- 能将数据库查询的细节和用户界面解耦。

选项需要硬编码在程序中，但对于表明、列名和 SQL 关键字来说，这么做还是很恰当的。对于数据值的选择往往是全量的字符串或者数字，但对于语法标识来说，可选择的范围很小。

21.5.5 找个可靠的人来帮你审查代码

找到瑕疵的最好方法就是再找一双眼睛一起来盯着看。你需要找个熟悉 SQL 注入的同事来帮助你一起检查代码。别让自大和骄傲阻止你做正确的事——现在承认你的代码有问题可能会让你感到窘迫，但你确定更愿意为黑客利用安全漏洞攻击网站而承担后果吗？

在检查代码是否包含 SQL 注入风险的时候，参考下面这几点。

(1) 找出所有使用了程序变量、字符串链接或者替换等方法组成的 SQL 语句。

(2) 跟踪在 SQL 语句中使用的动态内容的来源。找出所有的外部的输入，比如用户输入、文件、系统环境、网络服务、第三方代码，甚至于从数据库中获取的字符串。

(3) 假设任何外部内容都是潜在的威胁。对于不受信任的内容都要进行过滤、验证或者使用数组映射的方式来处理。

(4) 在将外部数据合并到 SQL 语句时，使用查询参数，或者用稳健的转义函数预先处理。

(5) 别忘了在存储过程的代码以及任何其他使用 SQL 动态查询语句的地方做同样的检查。

代码检查是找出 SQL 注入缺陷的最正确和经济的方法。你应该预留出相应的时间来做这件事，并且将其视为项目开发过程中一个必要的过程。当然，你可以帮助同事检查代码来回报他们为你所做的。

让用户输入内容，但永远别让用户输入代码。

第 22 章
伪键洁癖

重要的人不关心，关心的人不重要。

▶ 伯纳德·巴鲁克（在安排晚会餐桌席次时所说）

你的老板带着两份打印出来的报告过来找你，说："会计部的人说我们给出的这一季度报告和上季度报告有些差异。我正在看这两份报告，的确有差异，大部分最新的资产消失了。怎么回事？"

你看着这两份报告，发现这些差异看起来很眼熟。"不，每样东西都在那里。为了使所有的记录编号都是连续的，你让我整理过一次数据库。你说会计们由于数字之间的断档，一直在追问你中间那些不见了的资产是怎么回事。

"因此，我重新为一些记录编了号，然后把它们放在了原来的空行。现在没有断档了——从 1 到 12340 之间的每个数字都对应一个资产。所有的东西都在那里，只是有些改变了编号并且移到上面去了。是你告诉我这么做的。"

老板不住地摇头。"但这不是我想要的。会计人员是根据资产编号来跟踪设备的折旧状况的。每个设备的编号要在每个季度的报告中保持一致。除此之外，所有的资产编号都被打印并且贴在了对应的设备上。要花好几周的时间来重新为整个公司的设备贴新的标签。你能把所有的 ID 编号改回原来的吗？"

你想表现得自己很合作，因此转回电脑前开始工作，但你突然想到了一个新问题。"在我将这些资产 ID 改回原来的之后，这个月买的那些新资产怎么办？有些新资产使用了我重新编号之前就已在使用的编号。如果我把资产 ID 都改回原来的值，要怎么处理这些冲突？"

22.1　目标：整理数据

确实有这么一种人，他们会为一系列数据的断档而抓狂。

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
4	OPEN	ReConsider

一方面，bug_id 3 这一行确实发生过一些事情。为什么这个查询不返回这个 Bug？这条记录丢失了吗？那个 Bug 到底是什么？这个 Bug 是我们的一个重要客户提交的吗？我要为这条数据的丢失负责吗？

对于具有伪键洁癖这个反模式的那些人来说，他们的目的就是要解决这些麻烦的问题。这些人对数据的完整性问题负责，但通常来说，他们对数据库不够了解，或者不怎么相信数据库所生成的报表。

22.2 反模式：填充角落

大多数人对于断档的第一反应就是想要填补其中的空缺。对此，可能会有两种做法。

22.2.1 不按照顺序分配编号

你可能想要在插入新行时，通过遍历表，将找到的第一个未分配的主键编号分配给新行，来代替原来自动分配的伪主键机制。随着你不断地插入新行，断档就被填补起来了。

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
4	OPEN	ReConsider
3	NEW	Visual TurboBuilder

然而，为了找到第一个未被使用的值，你不得不执行一个不必要的自联合查询：

Neat-Freak/anti/lowest-value.sql

```
SELECT b1.bug_id + 1
FROM Bugs b1
LEFT OUTER JOIN Bugs AS b2 ON (b1.bug_id + 1 = b2.bug_id)
WHERE b2.bug_id IS NULL
ORDER BY b1.bug_id LIMIT 1;
```

在本书的前几章中，我们解释过在创建唯一标识的主键时，如果使用 SELECT MAX(bug_id)+1 这种查询语句，则会出现并发访问的问题。如果有两个程序同时想要找到最小的未使用值时，也会出现同样的问题，结果就是一个成功，另一个失败。况且，这个方法既低效，也容易导致问题。

22.2.2 为现有行重新编号

在某些情况下，你可能急着想要让所有的主键变得连续，而等着新行来填补空缺不够快。你可能会考虑更新现有行的主键值，让其变得连续，从而消除断档。通常这样的做法是找到主键最大的行，然后用最小的未被使用的值来更新它。比如，你可以将 4 更新为 3：

Neat-Freak/anti/renumber.sql

```
UPDATE Bugs SET bug_id = 3 WHERE bug_id = 4;
```

bug_id	status	product_name
1	NEW	Open RoundFile
2	FIXED	ReConsider
3	DUPLICATE	ReConsider

要完成这一步，你需要使用和前面一种方法中插入新行类似的方法。先找到一个未被使用的值，随后执行 UPDATE 语句来重新分配主键值。这两步都会引起并发访问的问题。而且，你需要重复执行很多次这样的操作，才能将较大的断档填满。

你必须同时更新所有引用了你重新分配了主键的行的子记录。如果你在定义外键时，使用了 ON UPDATE CASCADE 选项，这一步会变得很方便，但如果你没这么做，就必须先禁用约束，手动更新所有的子记录，然后恢复约束。这是一个费时费力且容易出错的操作，并且可能会影响到数据库的服务，正常人都会避免这么做的。

即使你完成了清理操作，“整洁”也是个“短命鬼”。当数据库生成一个新的伪键时，这个值是根据上次生成的值来计算的（即使上次生成的这条记录已经被删除了，也不会有任何影响），并不是按照现有记录中的最大值来计算的，这和一些数据库开发人员的假设相违背。假设你将最大的 bug_id 4 更新为最小的未被使用的值，然后消除了一个断档，下一次你使用默认的伪键生成器插入新行时，生成的伪键是 5，这就会在 4 的地方产生一个新的断档。

22.2.3 制造数据差异

Mitch Ratcliffe 说：“计算机是人类历史中最容易让你犯更多错误的发明……除了手枪和龙舌兰之外。”[①]

本章最开始的故事描述了重编号主键所存在的一些隐患。如果别的外部系统依赖于数据库中的主键来定义数据，那么你的更新操作就会使那个系统中的引用失效。

重用主键并不是一个好主意，因为断档往往是由于一些合理的删除或者回滚数据所造成的。比如，假设一个 account_id 为 789 的用户由于发送带有人身攻击性的邮件而被封号。产品策略

① *MIT Technology Review*，1992 年 4 月。

要求你删除这个账号，但如果你重用了主键，就会在某一个时间点将 789 分配给另一个用户。由于收件人可能在删除后的某一个时间点才打开这些邮件，他们还会投诉 789 这个账号的用户。尽管这个用户本身没有做错任何事情，但是他被分配了一个需要为此负责的编号。

别因为这些伪键看上去像是没用的而重新分配它们。

22.3 如何识别反模式

如果你发现团队成员问了如下几个问题，那么他们可能使用了“伪键洁癖”这个反模式。

- “在我回滚了一个插入操作后，要怎么重用那个自动生成的标识？”
  伪键一旦生成后不会回滚。如果非要回滚，RDBMS 就必须在一个事务的生命周期内生成一个伪键，而这在多个客户端并发地插入数据时，会导致竞争或者死锁。
- “bug_id 为 4 的这条记录怎么了？”
  这是对一个序列的主键中未使用的数字而感到焦虑的表现。
- “我要怎么找到第一个未使用的 ID？”
  要这么做的原因几乎肯定就是要重新分配 ID 了。
- “如果达到了数字标识的最大值怎么办？”
  这通常是重新使用未使用的 ID 的正当理由。

22.4 合理使用反模式

没有理由改变伪键的值，因为它的值本身并没有什么重要的意义。如果这个主键列有实际的意义，那么这就是一个*自然键*，而不是伪键。改变自然键的值并不奇怪。

22.5 解决方案：克服心里障碍

主键的值必须是唯一且非空的，因而你才能使用主键来唯一确定一行记录，但这是主键的唯一约束——它们不一定非得是连续值才能用来标记行。

22.5.1 定义行号

大多数伪键返回的数字看起来就像行号一样，因为它们就是*依次递增*的（每一个新返回的值都比前一个值大 1），但这只是由于伪键实现机制所造成的巧合而已。按照这样的方式生成主键值能比较方便地确保唯一性。

别把主键值和行号混为一谈。主键是用来标识表中记录的，而行号是用来标识查询结果集中记录的。查询结果集中的行号和主键没有一丁点关系，尤其是当你使用了 JOIN、GROUP BY 或者

ORDER BY 这些操作符的时候。

有很多使用行号的好理由，比如，返回一个查询结果集的子集。通常我们称之为分页，就像在网络搜索时的一页。要选择一个子集，你需要使用到实际的连续增长的行号，但和查询的形式无关。

SQL:2003 定义了包括 ROW_NUMBER()在内的一些窗口函数，返回一个查询结果集中一段连续的行。通常使用行号的作用是将查询结果限制在一个特定的范围内。

Neat-Freak/soln/row_number.sql

```
SELECT t1.* FROM
  (SELECT a.account_name, b.bug_id, b.summary,
     ROW_NUMBER() OVER (ORDER BY a.account_name, b.date_reported) AS rn
   FROM Accounts a JOIN Bugs b ON (a.account_id = b.reported_by)) AS t1
WHERE t1.rn BETWEEN 51 AND 100;
```

这些函数目前已经被很多主流数据库所支持，包括 Oracle、Microsoft SQL Server 2005、IBM DB2、PostgreSQL 8.4 和 Apache Derby。

MySQL、SQLite、Firebird 和 Informix 还不支持 SQL:2003 的窗口函数，但它们有一些独有的语法能用来处理本节所描述的问题。MySQL 和 SQLite 提供了一个 LIMIT 子句，Firebird 和 Informix 提供了使用 FIRST 和 SKIP 关键字的查询选项。

22.5.2 使用GUID

当然我们还可以生成随机伪键值，只要你不会重复使用任何数字。有些数据库提供全局唯一标识符（GUID）来达到这个目的。

GUID 是一个 128 位的伪随机数（通常使用 32 个十六进制字符表示）。由于 GUID 的定义及其所要实现的目的，它被设计成具有唯一性，因此你可以用其作为伪键。

下面这个例子用的是 Microsoft SQL Server 2005 的语法：

Neat-Freak/soln/uniqueidentifier-sql2005.sql

```
CREATE TABLE Bugs (
  bug_id UNIQUEIDENTIFIER DEFAULT NEWID(),
  -- . . .
);

INSERT INTO Bugs (bug_id, summary)
VALUES (DEFAULT, 'crashes when I save');
```

整数是不可再生资源吗?

和伪键洁癖反模式有关的误解是认为定量递增的伪键生成方式最终会用完整数集，未雨绸缪，不能浪费任何一个整数值。

乍看之下，这么说似乎有点道理。在数学中，整数是一个无限集合，但在数据库中，任何数据类型都只有有限数量的可选值，32 位整型最多只能表示 2^{32} 个不同的值。每生成了一个新的主键，离最后一个值就更近一步了。

我们可以算一下：如果你每天 24 小时不停地插入新的数据，平均每秒产生 1000 行记录，用完 32 位整型所能表示的这些数字总共需要 136 年。

如果这还不能满足你的需求，那就使用 64 位整型。这样，就算你每秒产生一百万条记录，也需要 584 542 年才能消耗完所有的整型数字。

所以，要把所有的整型用完几乎是不可能的！

这会生成如下形式的一行记录：

bug_id	summary
0xff19966f868b11d0b42d00c04fc964ff	Crashes when I save

GUID 相比传统的伪键生成方法来说，至少有如下两个优势。

- 可以在多个数据库服务器上并发地生成伪键，而不用担心生成同样的值。
- 没有人会再抱怨有断档——他们会忙于抱怨输入 32 个十六进制字符做主键。

下面这几点是 GUID 所带来的不便。

- GUID 的值太长，不便于输入。
- GUID 的值是随机的，因此找不到任何规则或者依靠最大值来判断哪一行是最新插入的。
- GUID 的存储需要 16 字节。这比传统的 4 字节整型伪键占用更多的空间，并且查询的速度更慢。

22.5.3 最主要的问题

虽然你已经清楚对伪键重新编号和一些相关方案可能会造成的问题，但你还有一个大问题没解决：怎么抵挡一个希望清理数据库中伪键断档的老板的请求？这是一个沟通方面的问题，而不是技术问题。无论如何，你需要让经理理解保护数据库中数据完整性的重要性。

- 向他解释 SQL 的技术。诚实往往是最好的方法。尊重并且理解这个要求之后的原因。比如说，你可以这样告诉经理：

“断档看上去的确很奇怪，但它们是无害的。在程序运行过程中，有些行被跳过、回滚或者删除都是很正常的。我们每次都为新插入的行分配一个新的值，而不是写代码来检测哪个老的值能安全使用。这样能使得开发更加简单，执行效率更高，并且能减少错误。”

- 清楚地表明开销。改变主键的值看上去是件小事，但你应该给出关于计算新值的工作量、写代码处理并测试重复值、级联修改整个数据库中的数据、调查对别的系统的影响以及训练用户和管理员使用新的存储过程等事情所需要花费时间的实际估算。

  大多数经理都会根据任务的成本来设定优先级，并且当他们面对真实的开销时，会撤销不合理的请求。

- 使用自然键。如果你的经理或者别的数据库用户坚持认为主键的值是有意义的，那就让主键变得有意义。别使用伪键——使用字符串或者有意义的数字。然后就很容易使用这些自然键所代表的意思来解释产生断档的原因。

  你还可以同时使用伪键和另一个被当成自然标识的属性列。在生成的报告中不显示伪键，从而隐藏会让读者感到不适的断档。

将伪键当做行的唯一性标识，但它们不是行号。

第 23 章

非礼勿视

在获得所有证据之前就下结论，是一个重大错误。

▶ 夏洛特·福尔摩斯

“我在你们的产品中找到了另一个 Bug。”电话那头说。

20 世纪 90 年代，当我还是一个 SQL RDBMS 的技术支持时，接到这个电话。我们有一个客户以总是提交一些荒谬的报告而出名。几乎所有他报告的问题最终都证明是他自己犯的小错误，而不是 Bug。

“Davis 先生，早上好。我们很荣幸能为你解决你所发现的问题。”我回答道：“你能告诉我发生了什么吗？”

“我向数据库发起了一个查询请求，可是什么都没返回。”Davis 先生很尖刻地说：“但我确信数据库里有数据，我用一个测试脚本验证过。”

“你的查询有问题吗？”我问道：“API 返回任何错误信息了吗？”

Davis 回应道：“为什么我要关心 API 函数的返回值？这个函数就应该执行我的 SQL 查询。如果它返回错误，就说明你的产品有 Bug。如果你的产品没有 Bug，就不会返回错误。我的工作不是解决你的 Bug。”

我目瞪口呆，但我必须让事实说话才能说服他。“好吧，我们来测试一下。将你的代码中的 SQL 查询语句复制粘贴到查询工具中，然后执行一下。返回什么？”我等待他回答。

“在 SELECT 附近有语法错误。”他停顿了一下，然后说：“你可以结束这个问题了。”然后他直接挂掉了电话。

Davis 先生是一家航空管制公司的开发人员，编写一些软件来记录国际航班的飞行数据。我们每周都要接到几个他的电话。

23.1 目标：写更少的代码

每个人都想写出优雅的代码。也就是说，我们想要用更少的代码来做更酷的事情。同时，我们所做的事情越酷，我们所要写的代码就越少，也越优雅。但如果我们不能让工作变得更酷，至少我们还有足够的理由来改进代码：用更少的代码来提高代码的优雅程度。

这仅仅只是表面上的理由，还有很多别的更靠谱的写出清晰代码的理由。

- 我们可以更快地写完一个程序的代码。
- 我们有更少的代码需要测试、写文档或审查。
- 更少的代码意味着更少的 Bug。

因此对于一个程序员来说，删除任何他们能删除的代码是他们的本能，尤其是那些不能让工作看起来更酷的代码。

23.2 反模式：无米之炊

“非礼勿视”对于开发人员来说通常有两种形式：第一，忽略数据库 API 的返回值；第二，和程序代码混在一起阅读那些分散的 SQL 语句片段。在这两种情况下，开发人员都会错过那些对他们修复错误非常有帮助的信息。

23.2.1 没有诊断的诊断

See-No-Evil/anti/no-check.php

```
<?php
❶ $pdo = new PDO("mysql:dbname=test;host=db.example.com",
     "dbuser", "dbpassword");
  $sql = "SELECT bug_id, summary, date_reported FROM Bugs
     WHERE assigned_to = ? AND status = ?";
❷ $stmt = $dbh->prepare($sql);
❸ $stmt->execute(array(1, "OPEN"));
❹ $bug = $stmt->fetch();
```

以上的代码非常简洁，但代码中有几处函数返回的状态值可能会引起问题，但如果你忽略了返回值，就永远不会知道怎么回事。

数据库 API 最有可能返回的错误就是在你建立和数据库之间的链接的时候，比如在代码❶处。你可能不小心打错了数据库的名字、服务器的地址、用户名密码，或者数据库服务本身挂了。在实例化一个 PDO 链接时，会抛出一个异常，可能会直接终止这个范例脚本的执行。

代码❷处调用的 `prepare()` 函数，可能会由于打字错误、不匹配的括号或者拼错了列名导致的语法错误而返回 `false`。代码❸处，`$stmt` 对象的成员函数 `execute()` 会产生一个致命错误，

因为 false 不是一个合法的对象。

```
PHP Fatal error:  Call to a member function execute() on a non-object
```

函数 execute()也会失败。比如，如果这条查询语句违反了某一个约束，或者超出了访问权限设定，这个函数也会返回 false。

在代码❹处，对函数 fetch()的调用，在有任何错误产生时，都会返回 false，比如无法链接到 RDBMS。

像 Davis 先生这样的程序员并不少见。他们可能会觉得对返回值进行检查或者对异常进行处理对于他们的代码来说没有任何意义，因为他们假设这些可能出错的情况是不可能发生的。同时，这些额外的代码都是重复的，并且让程序看起来很丑陋，而且难以阅读。它们的确一点也不酷。

但用户看不见代码，他们只能看见输出。当一个致命错误没有被处理时，用户就只能看到一个白屏（如图 23-1），或者是一个不完整的异常信息。当发生这种情况时，程序代码简短和清晰对用户来说，一点安慰作用都没有。

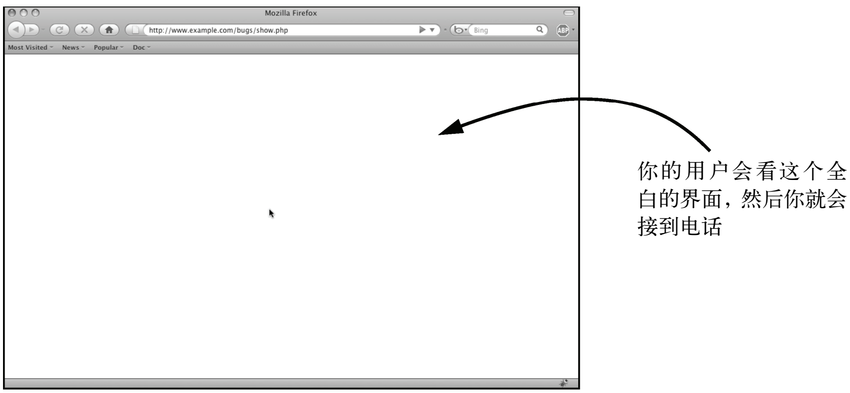


图 23-1　PHP 执行中的一个致命错误，导致白屏

23.2.2　字里行间

另一个常见的导致“非礼勿视”这个反模式的坏习惯是，盯着创建 SQL 查询字符串的代码调试。这么做对于调试来说比较困难，因为用程序逻辑、串连接和应用变量的额外内容建立之后，很难想象出最终拼出来的 SQL 查询语句到底是什么。这么调试代码就如同不看盒子上的图片直接开始玩拼图一样。

举个简单的例子，让我们看一个我经常听程序员问起的问题。下面的这段代码在断定用户查询一个特定的 Bug 而不是一个 Bug 集合后，会将 WHERE 子句连接到这个查询语句的后面，从而为其加上了条件查询的逻辑。

See-No-Evil/anti/white-space.php

```
<?php
$sql  = "SELECT * FROM Bugs";
if ($bug_id) {
    $sql .= "WHERE bug_id = " . intval($bug_id);
}
$stmt = $pdo->prepare($sql);
```

为什么这个简单的查询会返回错误？如果你看了连接后的$sql 变量里的值，真相就会浮出水面：

See-No-Evil/anti/white-space.sql

```
SELECT * FROM BugsWHERE bug_id = 1234
```

在 Bugs 和 WHERE 之间没有空格，造成了查询语句的语法错误。这条语句的意思变成了查询一张叫做 BugsWHERE 的表，但随后又跟着一串有语法错误的表达式。这段程序代码在连接两个字符串时没有加空格。

开发人员浪费了无数的时间和精力来调试这样简单的问题。通常只要看一眼创建完的 SQL 语句就能找到问题的症结，但开发人员更喜欢分析创建 SQL 语句的代码逻辑。

23.3　如何识别反模式

你可能认为，人工定位这些遗漏的错误处理代码很困难，你不见得需要这么做。很多现代的 IDE 都会在检测到没有处理返回值或者忽略了一个需要检测的异常[①]时，突出显示相应的代码。同时，你还可以根据下面的这些描述，推断出有人使用了“非礼勿视”这个反模式。

- “我的程序在我执行查询之后就崩溃了。”
  通常程序崩溃是因为查询失败，然后又试图以一种非法的方式使用返回的结果，诸如调用一个非对象的方法，或者引用一个空指针。
- “你能帮我找一下我的 SQL 查询中的错误吗？这是我的代码……”
  首先，看一下 SQL 语句，而不是生成它的代码。
- “我不会让错误处理弄乱了我的代码结构的。”
  一些计算机科学家推测在一个稳固的程序中，至少有 50%的代码是用来进行错误处理的。这看上去似乎很多，但想想在一个错误处理中所要包含的所有步骤：检查、分类、报告以及补救。对于所有的程序来说，实现错误处理都是很重要的。

① 需要检测的异常指的是在函数签名中附带的异常定义，因此使用者可以知道这个函数有可能会返回特定类型的异常。

23.4 合理使用反模式

当你真的不需要为错误负责的时候，的确可以忽略错误处理。比如，close()函数会关闭一个到数据库的链接，然后返回一个状态。但如果你的程序已经运行完成并准备退出了，那所有的链接资源无论如何都会被清空的，也就不需要关心 close()的返回值了。

在面向对象的语言中，异常允许你跟踪一个错误而不用处理它。任何调用你所提供的接口的代码，才是需要为这个异常负责的代码。因此，你可以直接将一个异常抛出并返回到调用栈的上一层。

23.5 解决方案：优雅地从错误中恢复

所有喜欢跳舞的人都知道，跳错舞步是不可避免的。优雅的秘诀就是弄明白怎么挽回。给自己一个了解错误产生原因的机会，然后就可以快速响应，在任何人注意到你出丑之前，神不知鬼不觉地回到应有的节奏上。

23.5.1 保持节奏

检查数据库 API 调用的返回状态和异常，是确保你不会跳错舞步的最佳方式。如下的代码对每一个会产生错误的调用都做了检查：

See-No-Evil/soln/check.php

```
<?php
try {
    $pdo = new PDO("mysql:dbname=test;host=localhost",
        "dbuser", "dbpassword");
❶ } catch (PDOException $e) {
    report_error($e->getMessage());
    return;
}

$sql = "SELECT bug_id, summary, date_reported FROM Bugs
    WHERE assigned_to = ? AND status = ?";

❷ if (($stmt = $pdo->prepare($sql)) === false) {
    $error = $pdo->errorInfo();
    report_error($error[2]);
    return;
}

❸ if ($stmt->execute(array(1, "OPEN")) === false) {
    $error = $stmt->errorInfo();
    report_error($error[2]);
```

```
        return;
    }

❹   if (($bug = $stmt->fetch()) === false) {
        $error = $stmt->errorInfo();
        report_error($error[2]);
        return;
    }
```

代码❶处在数据库链接失败时，捕获了其抛出的异常。当有问题的时候，其他的函数就会返回 false。

在检查了代码❷、❸和❹之后，你可以从数据库链接对象或者查询语句对象那里获得更多的错误信息。

23.5.2 回溯你的脚步

使用实际的 SQL 查询语句来进行调试也是很重要的，不能仅仅只是分析产生 SQL 查询语句的代码。很多简单的错误，诸如拼写错误、引号、括号不全都是非常明显的，即使它们在程序代码中让人费解、困惑。

- 使用一个变量来记录生成的 SQL 查询语句，而不是在调用 API 方法，传递参数的时候临时生成。这样做让你有机会在使用生成的 SQL 语句之前对其进行检查。
- 选择一个不是程序输出的地方输出 SQL 语句，比如日志文件、IDE 调试控制窗口，或者能显示调试输出的浏览器扩展①。
- 别将 SQL 语句当做 HTML 注释一起输出在页面中。因为任何人都能看页面源码。阅读这些 SQL 语句会让黑客了解很多关于数据库架构的信息。

使用一个对象关系映射（ORM）框架来透明地构建和执行 SQL 查询会使得调试更复杂。如果你不能获得 SQL 查询的上下文，又怎么观察并且调试问题呢？有些 ORM 通过将生成的 SQL 语句写入日志来解决这一矛盾。

最后一点，大多数数据库都提供了它们自己的日志机制，日志是记录在数据库服务器端而非程序客户端的。如果你不能在程序中记录 SQL 请求，仍可以在数据库服务器端监控这些请求的执行情况。

发现并解决代码中的问题已经很困难了，就别再盲目地干了。

---

① Firebug（http://getfirebug.com）就是个不错的扩展。

第 24 章 外交豁免权

人们讨厌改变。他们总是会说："我们一向如此。"而我尝试改变这一切，这就是为什么我墙上的钟是逆时针的。

► 葛丽丝 • 霍普少将

以前的一份工作曾给我上了重要的一课——关于使用软件工程最佳实践的重要性。那是在一次悲惨的事故之后，我开始接管一个重要的数据库程序。

我受聘于 Hewlett-Packard 公司，负责开发和维护一个基于 UNIX 的、使用 C 和 HP ALLBASE/SQL 编写的程序。面试我的经理和员工悲伤地告诉我，他们之前写这个程序的开发人员在一起交通事故中不幸去世了。他们部门中的其他人都不知道如何使用 UNIX，也不知道这个程序的任何细节。

我接手之后，发现之前的这个开发人员从来没有写过任何文档或者测试程序，也没有使用任何源代码版本控制程序，甚至连代码注释都没有。所有的源代码都放在同一个目录下，包括线上运行中系统的代码和开发阶段的代码，以及那些不再使用的代码。

这个项目在技术方面背负重重债务——使用捷径而不是最佳实践的后果。技术债务[①]（technical debt）会不断给项目带来风险和额外的工作，直到你重构、测试并为代码编写文档。

我总共花了六个月的时间来重新整理这些代码，并补全相关文档，因为我不得不花很多时间和精力为它的用户和后续开发做技术支持。然而，就程序本身而言，其实真的非常普通。

很显然，我没有办法让我的前任来帮忙加快项目进度。这个项目的经验证明了让技术债务失控所造成的影响有多严重。

24.1 目标：采用最佳实践

专业程序员总是努力地在其项目中使用好的软件工程习惯，比如下面几种。

① Ward Cunningham 在他关于 OOPSLA 1992 的报告中创造了这个单词。

- 将源代码使用版本控制工具管理起来，比如 SVN 或 Git。
- 为程序编写自动化单元测试脚本或者功能测试脚本。
- 编写文档、规格说明以及代码注释来记录程序的需求和实现机制。

使用最佳实践来开发软件所花费的时间是绝对有益的，因为这么做能减少很多不必要或者重复的工作。大多数资深开发人员都非常清楚地知道，如果为了方便起见而抛开这些实践方法，项目失败就是必然的。

24.2　反模式：将 SQL 视为二等公民

即使在接受并应用最佳实践的开发人员之中，也有一种思想趋向认为数据库代码是排除在这些实践方法之外的。我将这个反模式命名为“外交豁免权”，因为它假设程序开发的规则并不需要应用到数据库开发中去。

为什么开发人员会做出这样的决定？下面有几个可能的原因。

- 在有些公司里，软件工程师和数据库管理员的角色是分开的，DBA 通常同时和好几组程序员一起工作，因此可以说，DBA 对于任何一个项目组都不是全职的。DBA 被训练成一个参观者，并且不用和软件工程师负同样的责任。
- 使用在关系型数据库中的 SQL 语言和传统编程语言有很大不同。即使在程序中调用 SQL 语句的代码，它也看上去和原有代码在风格上格格不入。
- 高级 IDE 工具在程序编程语言中非常流行，其中编辑、测试和版本控制的功能非常便捷易学。但是数据库开发的工具就不这么高级了，或者使用范围不广。开发人员可以很简单地使用最佳实践来指导编码，但将这些应用到 SQL 上相比而言就显得很笨拙了。开发人员趋向于，宁可找点别的事情来做，也不会如此尝试。
- 在技术团队，对于数据库的通常认识和做法就是全由一个人来负责——DBA。因为 DBA 是唯一有权限访问数据库服务器的人，他通常被看做是一个活的资料库和版本控制系统。

数据库是一个程序的基础，和软件质量息息相关。你知道怎么写出高质量的代码，但可能会将程序构建在一个无法解决需求或者没人能看懂的数据库上。这样做的风险就是你在开发一个不得不最终放弃的程序。

24.3　如何识别反模式

你可能认为要证明没做什么很难，但并非每次都是如此。下面就是一些能说明问题的证据。

- “我们正在适应一个新的开发流程：一个轻量级的版本。”
  这里的“轻量级”通常意味着项目组想要跳过一些开发流程中的环节。其中一些可能是

合理的，但也可以理解为这是不遵循重要的最佳实践的借口。

- “新的版本控制系统的训练不需要 DBA 员工的参加，因为他们根本用不到。”
  排除一些技术团队的成员参与培训（或者可能都没有访问权限）确保了他们不会使用到这些工具。
- “我要怎么跟踪数据库中的表和列的使用情况？我们不清楚有些条目的作用，还有就是如果它们是孤立的，我们就考虑删除它们了。”
  你没有使用为数据库结构编写的项目文档，或者这份文档过期了，或者无法访问，或者根本就不存在。如果你不知道有些表和列存在的目的，也许它们对别人来说非常重要，你不能随意删除它们。
- “有没有什么工具能比较两个数据库结构，并给出两者的不同之处，然后生成一个将其中一个结构修改成另一个的脚本？”
  如果你没有遵循数据库结构变更部署的流程，它们就可能变得不同步，要将两个数据库的结构改成一致是一项很复杂的工程。

24.4 合理使用反模式

对于要多次使用的代码，我都会编写文档和进行测试，然后将代码置于版本控制之中，同时还有一些别的好习惯来处理这些代码。但我也会写一些即兴的代码，比如对一个 API 函数写的一次性测试代码以提醒我如何使用这个 API，或者回答一些用户提问的代码。

判断代码是否真是临时使用的好方法，就是在你使用完之后立刻删掉它们。如果你不能这么做，那这段代码可能就值得保留下来。同时，这也就意味着这段代码值得存进版本控制系统，并且至少需要写一些简明的注释，说明这段代码的作用以及使用方法。

24.5 解决方案：建立一个质量至上的文化

质量对于大多数程序员来说就是测试，但这仅仅是*质量控制*——整个流程中的一部分。软件工程的*质量保证*流程包含如下的三步。

(1) 清晰地定义项目需求，并且写成文档。
(2) 设计并实现一个解决方案来满足需求。
(3) 验证并测试解决方案符合需求。

你需要完成这三步操作才能确保正确的质量保证，虽然在某些软件开发方法中，不需要严格按照上面的这个顺序来执行。

通过遵循*编写文档*、*源码管理*和*测试*的最佳实践，你也可以在数据库开发中保证质量。

24.5.1 陈列A：编写文档

世上没有能够代替文档的代码。尽管一个资深的开发人员能够通过仔细的分析以及凭借自己丰富的经验来解读一段代码，但依旧是非常消耗体力的[①]。同时，代码也不能告诉你还有哪些问题没解决。

你应该将数据库的需求和实现也写成文档，就像对待程序代码那样。无论你是数据库的原始设计者，还是从别人那里接手的数据库，尽可能地使用下面的清单来检查文档的完整性。

实体关系图。整份数据库文档中最重要的部分，就是一张描述了数据库中所有表及表之间关系的ER图。本书有几章中使用了简单的ER图。更复杂一点的ER图包含了列、主键、索引和其他数据库对象。

不少绘图软件包含了ER图的标记。还有些工具能够通过SQL脚本或者运行中的数据库直接通过反向工程得到ER图。

当数据库过于复杂，表特别多时，要在一张ER图中将所有条目都表示出来会有点不切实际。在这种情况下，你应该将其拆分到多张图中。通常你可以使用“子组”这个符号，这样一来，ER图中即包含了足够的阅读信息，也不至于让人看得晕头转向。

表、列以及视图。你仍需要为数据库编写文档，因为ER图不是描述每张表、列及其他对象的目的和使用方法的正确形式。

对于表来说，需要描述一张表代表了哪种实体类型。比如 `Bugs`、`Products` 和 `Accounts` 很容易就能根据表名看出它们的意思，但对于 `BugStatus` 这种查询表，或者 `BugsProducts` 这种交叉表，又或者 `Comments` 这种依赖表来说，很难从表名上看出它们的作用。同时，还需要说明每张表预期存储多少行数据？你希望如何对这张表进行查询？表中有哪些索引？

每个列都有一个列名和对应的数据类型，但这些信息并不能说清楚这个列所代表的含义。比如说，哪些值对于这一列是有意义的（只有很少的情况下，取值范围是对应数据类型的所有可选值的全集）？这个列能否接受 `NULL`，为什么？有没有针对这一列的唯一性约束？如果有，为什么要有？

视图存储了对于一张或多张表的常用查询结果。为什么需要创建这样一个视图？哪个产品或者用户需要使用这个视图？这个视图是不是试图提取表之间的复杂关系？这个视图会不会让没有权限的用户查询一张需要授权的表的部分数据？这个视图是否可以更新？

---

① 如果代码是任何人都能读的，那为什么还叫代码呢？

关系。引用完整性约束表示了表和表之间的依赖关系，但可能并没有完整表达出你所设计的约束模型。比如，`Bugs.reported_by` 是非空的，但 `Bugs.assigned_to` 是可空的。这是不是意味着一个 Bug 可以在指派给任何人之前就被修复？如果不是，一个 Bug 的指派应该要遵循什么样的规则？

在某些情况下，可能存在一些隐式关系，但没有任何相关的约束。如果没有文档，别人很难知道存在这些关系。

触发器。数据验证、数据转换以及记录数据库变更，都是使用触发器的场景。你在触发器中实现了怎样的业务逻辑？这些都是需要记录在文档中的。

存储过程。要像 API 那样为存储过程编写文档。这个存储过程要解决什么问题？这个存储过程是否会改变数据？输入输出参数的类型和意义是什么？是否使用这个存储过程来替换一种查询请求，就能够解决某个性能瓶颈？使用这个存储过程是否会让没有权限的用户访问到需要权限的表？

SQL 安全。为应用程序指定了哪个数据库用户账号？每个用户都有哪些访问权限？你提供了哪些 SQL 任务，哪些用户可以使用这些任务？是否有些用户是为了特殊的任务所定义的，比如备份或报表？使用哪种系统安全级别规定，比如客户端必须通过 SSL 和 RDBMS 服务器端链接吗？使用何种机制检测和屏蔽非法的认证请求，比如暴力破解密码？有没有针对 SQL 注入做过一次完整的代码审查？

数据库基础设施：这些信息对运维和 DBA 的同事来说是最重要的，但开发人员最好也对此有些了解。使用哪个厂商的哪个版本的 RDBMS 服务？数据库服务器的主机名是什么？是否使用了多台数据库服务器、冗余备份、服务器集群、访问代理等？网络结构是什么样的？数据库服务器使用的是什么端口？客户端程序连接的时候有什么特殊的选项？数据库的用户密码是什么？数据库的备份方案是什么？

ORM。你的项目可能将一部分应由数据库处理的逻辑写在了程序代码中，作为基于 ORM 类的层的一部分。哪些业务逻辑是以这样的方式实现的？数据验证、数据转换、日志、缓存还是配置？

开发人员不喜欢维护工程文档。它们既难以编写，也很难保持实时更新，而且当你写了很多却少有人读的时候，会感到很失落。但即使是经验丰富或者极限编程的程序员也知道，他们可以不为程序的其他部分写文档，但一定要编写数据库部分的文档[①]。

① 比如，Jeff Atwood 和 Joel Spolsky 认为，除了数据库部分的文档，其他代码的文档基本没有意义。可以在 StackOverflow 上看到他们的讨论 http://blog.stackoverflow.com/2010/01/podcast-80/。

结构演化工具

版本管理工具管理了代码，但并没有管理数据库。Ruby on Rails 提供了一种技术叫做“迁移”，用来将版本控制应用到数据库实例的升级管理上。我们可以简单地来看一个升级的例子。

可以基于 Rails 的抽象类来编写一个脚本更新数据库，需写一个能一步完成数据库升级的函数，同时也需要写一个降级函数，能够反转升级函数的操作。

```
class AddHoursToBugs < ActiveRecord::Migration
  def self.up
    add_column :bugs, :hours, :decimal
  end

  def self.down
    remove_column :bugs, :hours
  end
end
```

Rails“迁移”工具会自动地创建一张表来记录当前数据库实例的多个版本。Rails 2.1 引入的修改让该系统更加的灵活，其随后的版本可能还会改变“迁移”的工作方式。

不断为数据库的每个结构变化创建新的迁移脚本会逐渐积累一系列的脚本，每一个脚本都能对数据库进行一步升级或者降级操作。如果你需要将数据库版本改为 5，只需要在运行“迁移”工具的时候指定参数。

```
$ rake db:migrate VERSION=5
```

可以参考“Rails 敏捷网站开发，第三版[RTH08]”或者访问 http://guides.rubyonrails.org/migrations.html 来对“迁移”工具进行更深入的学习。

大多数其他的网站开发框架，包括 PHP 的 Doctrine、Python 的 Django 以及微软的 ASP.NET，都支持类似于 Rails 的“迁移”这样的特性，可能集成在框架中，或者以相关项目的形式提供。

“迁移”使很多同步当前数据库实例和源码版本控制服务中指定版本结构的脏活能自动完成。但它们还不是完美的，只能处理一些简单类型的结构变更，而且从根本上说，它们在原有版本控制服务之外又建立了一个版本系统。

24.5.2 寻找证据：源代码版本控制

如果你的数据库服务器彻底挂掉了，要怎么重建一个数据库？跟踪数据库的一个复杂的升级过程的最有效途径是什么？要怎么回滚数据库的变更？

我们知道怎么使用版本控制系统来管理程序代码，解决软件开发中相似的一些问题。一个使

用版本控制的项目应该包含在现有程序被损坏时用以重建、部署该项目所需要的一切。版本控制也用来记录历史变更和增量备份，使得你能够回滚任意的修改。

对于数据库的代码，也能使用版本控制，从而获得和程序开发相似的效果。

Diplomatic_immunity/DatabaseTest.php

```
<?php
require_once "PHPUnit/Framework/TestCase.php";

class DatabaseTest extends PHPUnit_Framework_TestCase
{
    protected $pdo;

    public function setUp()
    {
        $this->pdo = new PDO("mysql:dbname=bugs", "testuser", "xxxxxx");
    }

    public function testTableFooExists()
    {
        $stmt = $this->pdo->query("SELECT COUNT(*) FROM Bugs");
        $err = $this->pdo->errorInfo();
        $this->assertType("object", $stmt, $err[2]);
        $this->assertEquals("PDOStatement", get_class($stmt));
    }

    public function testTableFooColumnBugIdExists()
    {
        $stmt = $this->pdo->query("SELECT COUNT(bug_id) FROM Bugs");
        $err = $this->pdo->errorInfo();
        $this->assertType("object", $stmt, $err[2]);
        $this->assertEquals("PDOStatement", get_class($stmt));
    }

    static public function main()
    {
        $suite  = new PHPUnit_Framework_TestSuite(__CLASS__);
        $result = PHPUnit_TextUI_TestRunner::run($suite);
    }

}

DatabaseTest::main();
```

你需要将和数据库开发相关的文件都提交到版本控制服务中去，包括如下的这些文件。

数据定义脚本。所有数据库都提供 CREATE TABLE 或别的定义数据库对象来执行 SQL 脚本。

触发器和存储过程。很多项目以数据库函数的形式为程序代码提供支持。你的程序可能离开这些函数根本无法工作，因此它们也算做你的项目代码的一部分。

初始数据。检查表可能包含一些数据，用以在任何用户输入新数据之前初始化数据库。你应该将这些初始数据保存起来，以备需要从项目源码重新构建数据库时使用。初始数据也叫做*种子数据*。

ER 图及文档。这些文件不是代码，但它们和代码、数据库需求、实现机制以及和程序的整合等联系密切。由于项目的升级依赖于数据库和程序的同时升级，你需要将这些文件也保持同步更新。需要确保这些文件切实描述了当前版本的设计。

DBA 脚本。大多数项目都有一系列的数据处理任务，这些任务都是在程序之外处理的，包括导入/导出数据、同步数据、生成报表、备份、验证数据、测试等。有些可能是用 SQL 脚本编写的，而非用一般的编程语言。

确保数据库代码文件是和使用当前数据库的程序代码相关联的。使用版本控制的部分好处就是，当从服务器端签出特定的版本、指定的日期或者不同的里程碑时，所得到的文件应该都能正常工作。你最好将数据库代码和程序代码放在同一个版本库中。

24.5.3 举证：测试

质量保证的最后一步就是质量控制——验证程序做了它应该做的。大多数专业的开发人员都很熟悉编写自动化测试脚本来验证程序代码的行为。测试的一个重要原则就是*隔离*，即同一时间点只测试系统的一个部分，如果存在缺陷，你可以尽可能缩小出错的范围。

我们在数据库测试中也借鉴这种隔离模块测试的方法，需要独立于程序代码对数据库结构及行为进行验证。

下面的例子展示了使用 PHP 单元测试框架写的一个单元测试[①]脚本：

你可以在验证数据库的时候参考如下的清单。

表、列和视图。你应该测试一下所希望存在的表、视图是否真的在数据库中存在。每次你使用新的表、视图或者列扩充数据库时，增加一个确认这些对象存在的新测试任务。你还可以使用*负面测试*，来验证一张表或者列在当前版本中是否真的删除了。

约束。这是另一个使用负面测试的地方。可以执行 `INSERT`、`UPDATE` 或者 `DELETE` 语句来验证约束是否有效，是否正确返回错误。比如，试着违反非空、唯一或者外键约束等。如果所执行

① 参考 http://www.phpunit.de/。确切地说，测试数据库的功能并不是严格意义上的*单元测试*，但还是可以使用这个工具来组织和使测试自动化。

的语句没有返回错误，那就意味着对应的约束有问题。你可以通过这样的测试尽早地找到更多的 Bug。

触发器。触发器也能进行强制约束。触发器能处理级联效果、转换数据、记录变更等。你应该通过执行一个语句来引发这个触发器测试这些场景，然后再进行一次查询来验证触发器是否按照你所期望的方式执行了。

存储过程。存储过程的测试最接近传统程序代码的单元测试。存储过程有输入参数，如果输入的参数无效则可能会抛出错误。存储过程内部的逻辑可能会有多个分支。存储过程可能会返回单个值，也可能返回一个查询结果集，这取决于输入的参数以及数据库中数据的状态。同时，存储过程也可能会受到数据库正在更新的副作用影响。你可以测试所有这些存储过程的特性。

初始数据。即使理论上完全空白的数据库也会需要一些初始数据，比如检查表中的记录。你可以使用一些查询语句进行查询来验证初始数据是否存在。

查询。程序代码依赖 SQL 查询。你可以在测试环境中执行一些查询操作来验证语法和结果。确认结果集中包含了所期望的列和数据类型，就像测试表和视图一样。

ORM 类。就像触发器一样，ORM 类也包含逻辑、验证、转换或者监控。你应该像对待其他的程序代码那样对基于 ORM 的数据库抽象代码进行测试。确认这些类在输入不同的参数时的行为如期望的那样，并且它们会拒绝接受非法参数。

如果这些测试中的任何一个没有通过，可能是因为程序在使用错误的数据库实例。总是要反复确认你所连接的数据库是否正确——最常见的错误其实就是连错数据库。如果必要，可以修改配置然后重新执行一遍测试。如果你确信链接是正确的，但需要修改数据库，那可以执行一个迁移脚本来同步这个数据库实例到程序所期望的版本。

24.5.4 例证：同时处理多个分支

在开发程序时，你可能需要同时操作多个版本，甚至可能在同一天就操作多个不同的版本。比如，你可能在当前部署的程序分支中修复了一个紧急 Bug，然后很快又回到主分支的长期开发中。

但程序所使用的数据库并没有版本控制。要在一眨眼的功夫内重新部署一个数据库是不现实的，即使你所使用的数据库非常敏捷和易于使用。

理想情况下，可以为每个开发、测试、分阶段进行或者部署的程序版本分支创建独立的数据库实例。同时，每个项目组内的开发人员也需要一个独立的数据库实例，从而在不影响整个团队中其他人开发进度的情况下，他们能够完成开发任务。

在程序配置项中增加选择数据库连接的字段，如此一来，无论你在哪个版本下工作，都可以在不修改代码的情况下指定一个数据库链接。

如今，每个 RDBMS 品牌的商业版和开源版本，都提供开发及测试的免费解决方案。使用诸如 VMware Workstation、Xen 和 VirtualBox 之类的平台虚拟化技术，每个程序员都可以以很小的代价运行一个服务器端基础设施的克隆版本，没有理由让程序员在一个和生产环境不一致的环境中进行开发和测试了。

数据库和程序都需要采用软件开发的最佳实践，包括文档、测试和版本控制。

第 25 章
魔豆

> 总会有解释，人类的每一个问题都会有一个众所周知的解决方案——无论是简洁的、可行的，还是错的。
>
> ▶ H. L. 门肯

“为什么加这么个小功能要花这么长时间？”经理指派你的团队扩展 Bug 跟踪系统，增加一个展示每个 Bug 有多少评论数量的功能。你的团队已经为此工作四周了。

在会议室里，你的组员们都羞于回答老板的问题。作为项目经理，你不得不回答：“我们在最开始的时候犯了几个错误，这个需求最初看起来实现很方便，后来我们意识到在程序中还有其他几个界面也需要展示评论数量。”

“设计这些界面要花四个礼拜？”经理问。

“这倒不是，这些界面只是一些 HTML，由于我们用的框架是将代码和展示分离的，这些界面的开发很简单。”你继续说，“但每次我们要往界面上增加一点新的条目，必须复制一份同样的代码到这个界面的后端代码中，用以获取数据。这意味着每个后端的类都需要进行一系列新的测试。”

“我们难道没有使用测试框架？”经理问，“增加一些新的测试用例要花多久？”

“编写测试不像编写代码那么容易，”另一个工程师犹豫地说，“我们还要为编写脚本构建测试数据。接着要在每次测试之前重新将数据载入测试数据库中。我们还要测试前端，每个新增的特性都要针对老功能的每一种组合进行测试。”

经理眼睛瞪得大大的，但你的同事继续说着：“我们现在对于前端已经有 600 个测试用例了，每一个测试都会创建一个新的浏览器模拟器。时间都花在执行这些测试上了。”他耸耸肩，“我们对此无能为力。”

经理深吸了一口气，说：“好吧……你说的我并不全都理解，我只是想要知道为什么加这么一个简单的功能会变得如此复杂。你们所用的面向对象框架难道不支持快速、简单地添加新功能吗？”

好问题！

25.1　目标：简化 MVC 的模型

Web 程序框架使得往程序中添加新功能变得更简单、快速。在软件项目中，最大的成本就是开发时间。因此，我们所减少的任何一点开发时间，都能使得软件开发的成本降低。

Robert L. Glass 认为：“80%的软件工作是智力活动。相当大的比例是创造性的活动。很少是文书性的工作。”①

有一种方法有助于我们在软件开发过程中思考，那就是采用设计模式的术语和习俗。当我们说单例模式、外观模式或者工厂模式的时候，项目组内的其他开发人员都知道我们在说些什么。这样做节省了很多时间。

在任何项目中，大多数的代码都是重复的——几乎都长得一样。框架通过提供可重用的组件和代码生成工具帮助我们提升编写代码的速度，做到少写代码也能开发出可以工作的软件。

当使用模型–视图–控制器（MVC）架构的时候，我们就是同时在使用设计模式和软件框架。这是一个拆分程序逻辑的技术，就如图 25-1 所展示的那样。

- 控制器接收用户的输入，定义一些程序需要完成的响应逻辑，再委托合适的模型执行操作，然后将结果返回给视图。
- 模型处理所有其他的事情，它们是程序的核心，包括输入验证、业务逻辑和数据库交互。
- 视图处理在用户界面展示信息。

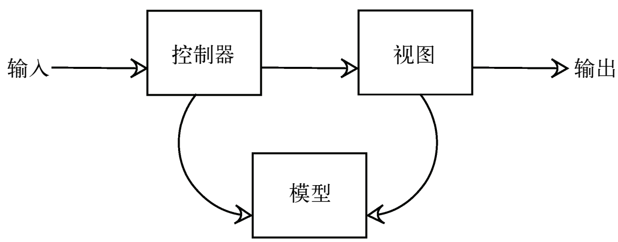


图 25-1　模型–视图–控制器（MVC）

理解控制器和视图的行为很容易，但是模型的目的就比较模糊。在软件开发人员的社区中常

① *Facts and Fallacies of Software Engineering*[Gla92]。

讨论的问题便是，为了减少软件设计的复杂度，如何简化和抽象模型。但是通常这个目标会导致他们过度简化模型，以至于把它仅仅视作一个数据访问对象。

25.2 反模式：模型仅仅是活动记录

在简单的程序中，你不需要在模型中定义很多逻辑。相对而言，模型更像是数据库中的某个表的映射对象，也是一种*对象-关系映射*（ORM）。你需要这个对象做的所有事情是知道如何往表中插入新行、读取一行，以及更新和删除自己——基本的 CRUD[①]操作。

Martin Fowler 将这种映射关系概括为一种设计模式，叫做*活动记录*[②]。活动记录模式是一种数据访问模式。其方法是将一个类与数据库中的一张表或者一个视图相关联。你可以调用这个类的 `find()`方法返回一个关联到这张表或者视图的某一行的类对象实例。你还可以使用这个类的构造器来创建一个新行。调用 `save()`方法执行插入或者更新现有的行。

Magic-Beans/anti/doctrine.php

```
<?php
$bugsTable = Doctrine_Core::getTable('Bugs');
$bugsTable->find(1234);

$bug = new Bugs();
$bug->summary = "Crashes when I save";
$bug->save();
```

自 2004 年开始，Ruby on Rails 使活动记录模式在 Web 程序开发框架中流行起来，现在大多数的 Web 程序框架使用这种模式作为数据访问对象（DAO）。使用活动记录模式并没有什么错，这是一个很好的模式，提供了简单访问表中特定行的接口。我们所要说明的反模式是在 MVC 程序中，所有模型类都继承自同一个活动记录基类这一习惯做法。这是一个“金锤子”反模式的例子：如果你的唯一工具是一把锤子，那么就会将每个东西都看做钉子。

所有能简化软件设计的做法都很吸引人。如果我们愿意牺牲一些灵活性，就能让工作更简单，而且，如果灵活性从一开始就不重要，那就更好了。

但这是个童话故事，就像《杰克与魔豆》一样。杰克相信当他睡着的时候，他的魔豆会长成一个巨大的豆茎。在杰克的故事中的确就是这么发生的，但我们不可能都这么幸运。让我们来看看使用魔豆反模式的后果。

① Create, Read, Update Delete，增删改查操作。——译者注

② 参考 *Patterns of Enterprise Application Architecture* 中“Active Record”一章。

抽象泄露

Joel Spolsky 在 2002 年提出了“抽象泄露”这个词。*抽象简化了一些技术的内部工作原理并且让其更加方便调用。但当由于需要更高效的生产效率而不得不了解内部细节的时候，就称之为抽象泄露。

在 MVC 架构中，把活动记录模式作为模型层来使用就是抽象泄露的例子。在非常简单的项目中，活动记录模式就像魔法一样神奇。但如果你想要在所有的数据库访问上都使用这个模式，你就会发现很多诸如 `JOIN` 或者 `GROUP BY` 的这些在 SQL 中很简单的操作，在活动记录模式中变得很可怕。

有些框架尝试扩展活动记录模式来支持更大规模的 SQL 语句。但是框架暴露的使用 SQL 内部接口越多，你就会越觉得不如直接使用 SQL 来得方便。

抽象模式因此不再能隐藏它的秘密，就像绿野仙踪里的托托发现精灵不过是藏在窗帘后的普通人一样。

* 参考《抽象泄露法则》[Spo02]

25.2.1 活动记录模式连接程序模型和数据库结构

活动记录模式是一种简单的模式，因为一个普通的活动记录类只能表示数据库中一张表或者一个视图。活动记录对象中的每一个字段对应于相关表中的一列。如果你有 16 张表，就要定义 16 个模型子类。

这意味着，如果需要重构数据库来表示一个新数据结构，你的模型类就需要跟着改变，同时，任何使用这些模型类的代码也都需要改变。还有，如果增加了一个控制器来处理新的程序界面，你也需要重复这些查询模型的代码。

25.2.2 活动记录模式暴露了CRUD系列函数

接下来你需要面临的问题就是，其他使用模型类的程序员会无视你设定的使用方法，他们会直接使用 CRUD 函数来更新数据。

比如，你可能在 Bug 模型中有一个叫做 `assignUser()`的方法解决每次更新 Bug 之后发送一封邮件给相关工程师的需求。

Magic-Beans/anti/crud.php

```
<?php
class CustomBugs extends BaseBugs
{
```

```
  public function assignUser(Accounts $a)
  {
    $this->assigned_to = $a->account_id;
    $this->save();
    mail($a->email, "Assigned bug",
      "You are now responsible for bug #{$this->bug_id}.");
  }
}
```

然而，另一个编程人员忽略了你的方法，他手动地分配了这个 Bug 却没有发送邮件。

Magic-Beans/anti/crud.php

```
$bugsTable = Doctrine_Core::getTable('Bugs');
$bugsTable->find(1234);
$bug->assigned_to = $user->account_id;
$bug->save();
```

你的需求是无论何时对一个任务进行变更操作，都要有一封邮件提醒。而这种模型设计允许略过这一步。将活动记录类的 CRUD 方法暴露给驱动模型类是否真的有意义呢？要如何阻止那些使用这些方法的程序员的不合理调用？如何在编写项目文档和具体代码实现的时候，将这些活动记录的接口从你的模型类中排除呢？

25.2.3 活动记录模式支持弱域模型

另一个非常值得关注的问题，就是一个活动记录模型通常除了 CRUD 方法之外，没有别的行为。很多程序员扩展了这个基本的活动记录类，却不为这个模型所需处理的业务逻辑添加新的方法。

将模型简单地当成数据访问对象，鼓励开发人员将业务逻辑放在模型类之外去实现，通常这些逻辑就会被拆分到多个控制类中，从而减少了模型本身的内聚行为。Martin Fowler 在他的博客里称这种反模式为*弱域模型*[①]。比如说，你可能有一系列独立的活动记录类，它们分别关联到 `Bugs`、`Accounts` 和 `Products` 表，但在很多地方你都是同时需要这三张表中的数据的。

让我们看一段 Bug 跟踪程序中的代码，其简单地实现了 Bug 指派、数据输入、Bug 显示和搜索的工作。它使用的 PHP 框架叫做 Doctrine，这个框架提供简单的活动记录接口，并且使用 Zend 框架来实现 MVC 架构。

Magic-Beans/anti/anemic.php

```
<?php
class AdminController extends Zend_Controller_Action
{
  public function assignAction()
```

① http://www.martinfowler.com/bliki/AnemicDomainModel.html。

```
    {
      $bugsTable = Doctrine_Core::getTable("Bugs");
      $bug = $bugsTable->find($_POST["bug_id"]);
      $bug->Products[] = $_POST["product_id"];
      $bug->assigned_to = $_POST["user_assigned_to"];
      $bug->save();
    }
  }

  class BugController extends Zend_Controller_Action
  {
    public function enterAction()
    {
      $bug = new Bugs();
      $bug->summary = $_POST["summary"];
      $bug->description = $_POST["summary"];
      $bug->status = "NEW";

      $accountsTable = Doctrine_Core::getTable("Accounts");
      $auth = Zend_Auth::getInstance();
      if ($auth && $auth->hasIdentity()) {
        $bug->reported_by = $auth->getIdentity();
      }
      $bug->save();
    }

    public function displayAction()
    {
      $bugsTable = Doctrine_Core::getTable("Bugs");
      $this->view->bug = $bugsTable->find($_GET["bug_id"]);

      $accountsTable = Doctrine_Core::getTable("Accounts");
      $this->view->reportedBy = $accountsTable->find($bug->reported_by);
      $this->view->assignedTo = $accountsTable->find($bug->assigned_to);
      $this->view->verifiedBy = $accountsTable->find($bug->verified_by);

      $productsTable = Doctrine_Core::getTable("Products");
      $this->view->products = $bug->Products;
    }
  }

  class SearchController extends Zend_Controller_Action
  {
    public function bugsAction()
    {
      $q = Doctrine_Query::create()
        ->from("Bugs b")
        ->join("b.Products p")
        ->where("b.status = ?", $_GET["status"])
        ->andWhere("MATCH(b.summary, b.description) AGAINST (?)", $_GET["search"]);
```

```
    $this->view->searchResults = $q->fetchArray();
  }
}
```

使用活动记录的控制器代码逐渐变成组织程序逻辑的技术手段。如果数据库的结构或者程序的期望行为改变了，你就需要更新代码中的很多地方。同时，如果增加了一个控制器，即使这个控制器对模型对象的查询逻辑和其他的控制器中的实现很类似，你也要编写新的代码来处理。

类交互图（图 25-2）很混乱而且难以阅读，当增加了新的控制器和 DAO 类时，它只会变得更糟糕。这是一个很强烈的信号——这些同时使用不同模型的代码在多个控制器复制来复制去，你需要使用另一种方法来简化和压缩程序。

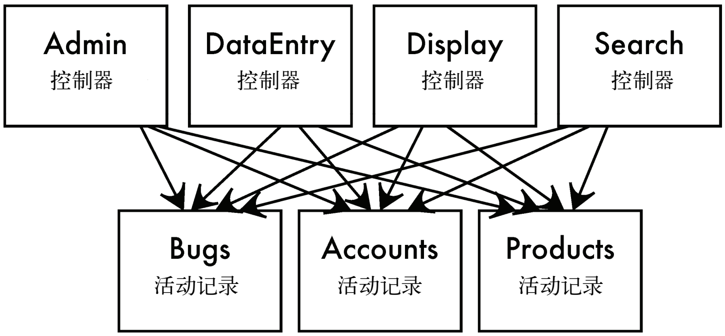


图 25-2　使用魔豆造成混乱

25.2.4　魔豆难以进行单元测试

使用了魔豆反模式，你会发现测试 MVC 的每一层都变得更加困难。

- 测试模型：由于将模型类等同于活动记录类，你无法将数据访问与模型行为分开测试。要测试这些模型，你就必须连上真实的数据库执行查询。
  很多人使用数据库固定工具。数据库固定工具将数据载入到测试数据库中，来确保每个测试使用的是同样的标准数据。这样复杂的步骤使得测试模型的过程变得缓慢且容易出错，就和请求真实数据库进行测试一样麻烦。
- 测试视图：测试视图包括将视图呈现成 HTML 内容并解析其结果，来验证由模型提供的动态 HTML 条目确实出现在了输出中。即使你所使用的框架简化了测试脚本中的判定过程，框架还是要执行复杂且耗时的代码来呈现及解析 HTML 中指定的条目。
- 测试控制器：你将发现测试控制器也变得很复杂，因为模型就是导致同样的代码在多个控制器中重复出现的数据访问对象，所有的代码都需要测试。
  要测试控制器，你需要模拟一个 HTTP 请求。Web 程序的输出是一个 HTTP 响应包的包头和包体。要验证这个测试的结果，就不得不拆分由控制器返回的 HTTP 响应的内容。这

需要很多的代码来测试业务逻辑，导致测试进行得很慢。

如果你能够将业务逻辑和数据库访问以及展示层拆分开，将有助于达成 MVC 的目标，同时也会让测试变得更简单。

25.3 如何识别反模式

下面的线索可能意味着你用了魔豆反模式。

- “我要怎么传递一个自定义的 SQL 查询给模型？”
  这个问题表示了你正在使用一个数据库访问类做为模型类。你不应该将 SQL 查询语句传递给模型对象——模型类应该囊括了所有它需要的查询。
- “我应该复制复杂的模型查询到我所有的控制器内，还是在一个抽象控制器内实现这些代码？”
  这两种方案都无法给你带来简单性或者稳定性。你应该将复杂的查询代码写在模型类里面，作为模型类的接口暴露出来。这样，就遵循了“不要重复自己（DRY[①]）”的原则，并且模型使用起来更简单。
- “我不得不写更多的数据库固定工具来对模型进行单元测试。”
  如果你正在使用数据库固定工具，就是说正在测试数据库访问而不是业务逻辑。你应该将对模型的单元测试和数据库分离开。

25.4 合理使用反模式

本质上说，活动记录模式并没有什么错，对于简单的 CRUD 操作来说，这是一个很方便的模式。在大多数程序中，总有一些情况只需要简单的数据访问对象，来提供一些简单的对于表的行操作。此时，你可将模型和 DAO 视作同一个东西。

另一个使用活动记录的好地方是原型开发。当快速编码比写代码的可测试性和可维护性还重要的时候，捷径是关键。在早期的工作中展示一个原型来获得积极的反馈，通常是提炼项目的一个好方法。任何你能加速原型开发的方法在这些情况下都是很有帮助的，使用一个简单的程序框架也很有用。

此外，你必须要确认你留有一定的时间来重构代码，从而偿还在原型开发阶段的技术债务。

25.5 解决方案：模型包含活动记录

控制器处理程序输入，视图处理程序输出，两者的任务都相对简单且定义清楚。框架是帮助

① DRY 出现在 *The Pragmatic Programmer*[HT00]，由 Andy Hunt 和 Dave Thomas 所著。

你将这些东西快速融合在一起的最好方法。但是对于框架来说，很难给模型提供一个“通吃”的解决方案，因为模型包含了面向对象设计中的其余部分。

这是你确实需要仔细考虑的部分，关于你的程序中的对象是什么，以及这些对象包含什么数据和行为。还记得 Robert L. Glass 评估软件开发中的主要工作是智力活动和创造力吗？

25.5.1 领会模型的意义

所幸的是，在面向对象设计领域，有很多格言警句能指导你。比如，Graig Larman 的《UML 和模式应用》（*Applying UML and Patterns*[Lar04]）一书，描述了很多指导方针，被称为通用职责软件分配模式（GRASP）。其中的一些指导原则和分离模型和数据访问对象尤其相关。

信息专家

一个对象应该有所有需要的数据来满足它所负责的操作。由于程序中的一些操作可能会包含多个表（或者没有表），并且活动记录只适用于一次操作一张表，我们需要另一个类来聚合多个数据库访问对象，并利用这些对象进行组合操作。

模型和活动记录之类的 DAO 之间的关系应该是 HAS-A（聚合）而不是 IS-A（继承）。大多数依赖于活动记录模式的框架都使用 IS-A 的解决方案。如果你的模型使用 DAO 而不是直接从 DAO 类继承，那么你就能将这个模型设计成包含所有的数据和代码来表达它代表的领域的形式——即使需要用多张表来表示。

创造者

一个模型如何维护数据库应该是它的内部实现细节，一个聚集了一系列 DAO 的领域模型应该负责创建这些对象。

程序的控制器和视图应该使用领域模型的接口，而不用关心这个模型使用哪种数据库交互方式对数据进行存取。这使得日后修改数据库查询变得更加容易，只需要修改一个地方。

低耦合

解耦程序中的逻辑区块是非常重要的，这么做能够让你更加灵活地改变某一个类的实现，同时又不影响这个类的调用方式。程序的需求是无法简化的，一些复杂的逻辑无法在程序中舍弃，但你可以对在何处实现这些复杂的逻辑做出正确的选择。

高内聚

领域模型的接口应该反映出它所期望的调用方式，而不是数据库物理结构或者 CRUD 操作。通常的活动记录模式的接口，如 `find()`、`first()`、`insert()`或者 `save()`，并没有给出多少对

软件需求实现方面的信息。而类似于 assignUser()的方法则更加具有说明性，并且控制器代码也能变得更容易理解。

将模型类和它所使用的 DAO 解耦后，你甚至可以为同一个 DAO 设计多个模型类。这比将所有关于给定表的相关工作都合并到单一展开的活动记录类中要好很多。

25.5.2　将领域模型应用到实际工作中

在《领域驱动设计：软件核心复杂性应对之道》（*Domain-Driven Design：Tackling Complexity in the Heart of Software*[Eva 03]）一书中，Eric Evans 介绍了一个更好地解决方案：领域模型。

在最初的 MVC 概念中（而不是武断的软件设计），一个模型所表示的是程序中某一领域的面向对象的表现手段，也就是说，程序中的业务逻辑和所需要的数据。模型是实现程序业务逻辑的地方，将数据存储在数据库中是模型的内部实现细节。

既然我们让模型设计围绕着程序的逻辑，而不是数据库层面，就可以将数据库操作完全隐藏在模型类里面。看一下可以重构我们早先的例子的几处地方：

Magic-Beans/soln/domainmodel.php

```
<?php

class BugReport
{
  protected $bugsTable;
  protected $accountsTable;
  protected $productsTable;

  public function __construct()
  {
    $this->bugsTable = Doctrine_Core::getTable("Bugs");
    $this->accountsTable = Doctrine_Core::getTable("Accounts");
    $this->productsTable = Doctrine_Core::getTable("Products");
  }

  public function create($summary, $description, $reportedBy)
  {
    $bug = new Bugs();
    $bug->summary = $summary
    $bug->description = $description
    $bug->status = "NEW";
    $bug->reported_by = $reportedBy;
    $bug->save();
  }

  public function assignUser($bugId, $assignedTo)
```

```
  {
    $bug = $bugsTable->find($bugId);
    $bug->assigned_to = $assignedTo"];
    $bug->save();
  }
  public function get($bugId)
  {
    return $bugsTable->find($bugId);
  }

  public function search($status, $searchString)
  {
    $q = Doctrine_Query::create()
      ->from("Bugs b")
      ->join("b.Products p")
      ->where("b.status = ?", $status)
      ->andWhere("MATCH(b.summary, b.description) AGAINST (?)", $searchString]);
    return $q->fetchArray();
  }
}


class AdminController extends Zend_Controller_Action
{
  public function assignAction()
  {
    $this->bugReport->assignUser(
      $this->_getParam("bug"),
      $this->_getParam("user"));
  }
}


class BugController extends Zend_Controller_Action
{
  public function enterAction()
  {
    $auth = Zend_Auth::getInstance();
    if ($auth && $auth->hasIdentity()) {
      $identity = $auth->getIdentity();
    }
    $this->bugReport->create(
      $this->_getParam("summary"),
      $this->_getParam("description"),
      $identity);
  }


  public function displayAction()
```

```
  {
    $this->view->bug = $this->bugReport->get(
      $this->_getParam("bug"));
  }
}
class SearchController extends Zend_Controller_Action
{
  public function bugsAction()
  {
    $this->view->searchResults = $this->bugReport->search(
      $this->_getParam("status", "OPEN"),
      $this->_getParam("search"));
  }
}
```

你可能注意到了几处改进。

- 类交互图（图 25-3）变得更加简单且易读了，象征着各个类之间的解耦。
- 通过解耦模型接口和底层的数据库结构，我们减少且简化了控制器的代码。
- 模型类创建对象和一个或者多个表进行交互，控制器不需要知道哪张表被引用了。
- 模型类封装、隐藏了数据库查询，控制器只关心用户的输入，然后通过调用模型的 API 来执行更高层次的任务。
- 某些情况下，一个查询太复杂而无法简单地使用一个 DAO，编写自定义的 SQL 就变得很必要。SQL 安全地包含在模型类里时，直白地使用它看上去不那么可怕。

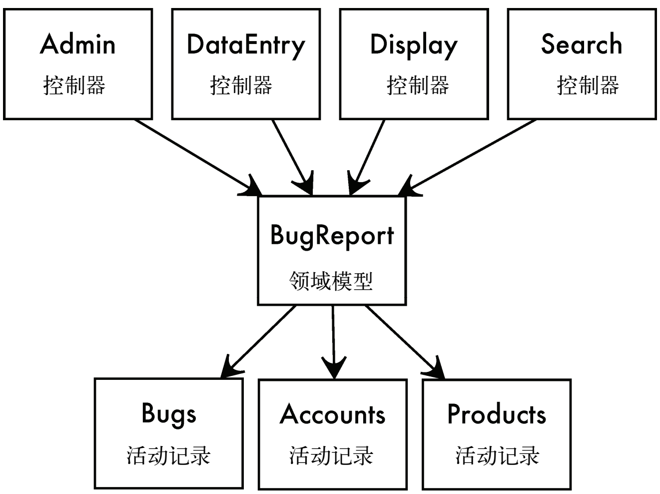


图 25-3　通过解耦减少混乱

25.5.3　测试简单对象

理想情况下，你可以不用连接到真实的数据库来测试模型。如果将模型和 DAO 解耦，那么

可以模拟 DAO 来帮助对模型进行单元测试。

同样，你可以像其他面向对象的测试那样测试领域模型的接口：调用对象的方法，然后验证这个方法的返回值。这比模拟 HTTP 请求来填充一个控制器，并且解析 HTTP 的响应要快得多，容易得多。

你还是需要模拟 HTTP 请求来测试控制器，但由于控制器的代码变得更简单，所以不需要测试很多逻辑分支。

如果将模型和控制器、数据组件分离，就可以对所有这些类都进行更简单的、更独立的单元测试。这能帮助你更容易地定位缺陷。这可就是单元测试的目的所在啊！

25.5.4　回到地球

你可以在任何软件开发框架中更有效地使用数据访问对象，即使这个框架鼓励使用魔豆反模式。然而，那些不知道如何使用面向对象设计原则的开发人员，注定会写出意大利面条式的代码。

本章所提及和描述的领域模型的基本概念，能帮助你选择最好的设计来支持测试和代码维护，最终会达到高效开发基于数据库的程序的目的。

将模型和表解耦。

附录A 规范化规则

年轻人，在数学里，你们并不理解事情，你们只是去习惯。

▶ 约翰·冯·诺伊曼

设计关系型数据库的过程既不靠臆断，也不神秘。你可以使用定义好的一系列规则来设计数据存储策略，从而避免冗余并防止应用程序出错，就像本书之前提到的 poka-yoke 方法一样。你可能听说过类似的一些概念，比如“防御设计”或者“尽早暴露错误”等。

规范化规则并不复杂，但很隐晦。开发人员通常会误解它们的原理，可能是因为他们将规则想得过于复杂了。

另一种可能性就是开发人员本身抗拒遵循规则。规则是那些具有创新精神的程序员所鄙视的东西，也是自由的对立面。

软件开发人员需要不断在简单和灵活之间进行取舍。你可以做很多重新发明轮子的工作和开发自定义的数据管理软件，或者通过遵循相关的设计，利用已有的知识和技术。

本书中我根据那些反模式的优点（或者缺点）来描述它们，来避免过于学术或者理论性。但在这个附录中，我们将看到理论也可以变得很实用。

A.1 关系是什么

“关系”这个术语并不是指表和表之间的关系。它指表自身，或者更进一步，是指表中列之间的关系。某种程度上来说，它也同时表示这两种关系。

数学家将*关系*定义为，两个不同数据域上的值的集合通过一定的条件得到一个所有可能组合的子集。

比如，一个包含所有棒球队名字的集合，和一个包含所有城市的集合。将每个城市和球队的组合都列出来，这个列表可以很长很长。但我们只关注这个列表的一个子集：球队和其所属

城市的组合。有效的组合包括 Chicago/White Sox、Chicago/Cubs 或者 Boston/Red Sox，但没有 Miami/Red Sox。

“关系”这个词有两种用法：作为一种规则（“城市指某一支队伍所属的城市”），或者作为过滤子集的规则。在 SQL 中，我们可以将这个子集存储在一张有两列的表中，每一行对应一个组合。

当然，关系并不局限于只有两列，可以将任意数量的数据域，每个占一列，合并在一起组成一个关系。同时，数据域可以是 32 位整型数的集合，也可以是固定长度字符串的集合。

在对表进行规范化处理之前，需要确认它们的关系是合适的，它们必须满足一些条件。

A.1.1　行之间没有上下顺序

在 SQL 中，查询返回的结果的顺序是不定的，除非使用 ORDER BY 指定排序规则。当然，除了顺序，结果集中的数据总是一致的。

A.1.2　列之间没有左右顺序

无论是让 Steven 测试 Open RoundFile 这个项目的第 1234 号 Bug，还是我们需要知道 Open RoundFile 的第 1234 号 Bug 是否能让 Steven 来验证，最终得到的查询结果应该是一样的。

这和第 19 章的反模式相关，那时我们使用列的顺序而不是列名。

A.1.3　重复行是不允许的

对于一个事实，进行更多的证明也不会让它变得更正确。给定一个棒球队的队名，数据就能指出它所在的城市是哪个，我们可以说城市依赖于队名。

要阻止重复记录，我们必须能区别两行数据，并且能定位指定的行。在 SQL 中，我们对列或者列的集合使用主键约束来确保这一点，而不管是否需要保证记录唯一性。

在其他非主键列里可能还是存在重复——波士顿有两支球队——但整体来看每一行是不重复的，因为这两支队伍的队名不同。

A.1.4　每一列只有一种类型，每一行只有一个值

关系头定义了列的名字和数据类型。每一行数据拥有的列和头中的定义必须匹配，且每一列的意义在所有行中必须一致。

第 6 章的反模式有两处违反这个规则的地方。首先，EAV 的表让每个实例的模型都可以自定义属性集，因此，实体的结构和任何头定义都不一样。

其次，EAV 的 `attr_value` 列包含了所有实体的属性，包括 Bug 的报告日期、Bug 的状态、Bug 被指派给哪个账户等。1234 这个值可能对于两个不同的属性来说都是合法的，但又完全具有不同的含义。

第 7 章中的反模式也破坏了这条规则，由于 1234 这个值可能会引用任意多个父表中的主键，因此无法断定不同两行中的 1234 是否具有相同的含义。

A.1.5 行没有隐藏组件

列中存储的是数据的值，不包含物理存储标示，诸如行 ID 或者对象 ID。在第 22 章中，我们知道主键是唯一的，但并不是实际的行号。

有些数据库绕开了这条规则，提供了访问数据库内部存储细节的扩展 SQL 语法(比如，Oracle 的 `ROWNUM` 伪列，或者 PostgreSQL 中的 `OID` 列)。然而，这些数据并不属于关系结构。

A.2 规范化的神话

很难再找出一个像规范化这种即使有精确定义依旧被广泛误解的主题。实际生活中，你肯定遇到过相信下面这些错误理念的开发人员。

- “规范化让数据库更慢。不规范让数据库更快。”

  错！的确，应用了规范化之后，查询时可能需要使用 `JOIN` 从多张分开的表中获取数据，而不规范的数据能够避免这些 `JOIN`。

  比如，在第 2 章中使用逗号分割的列表来获取 Bug 相关的产品。但如果我们同时还需要根据指定的产品获取所有相关的Bug呢？非规范化通常能简化或者提升某种类型的查询，但应用在别的查询类型上时开销就很大。

  非规范化也有合理使用的场景，但是在决定使用它时，应该先将数据库设计成标准的形式。第 13 章中的 MENTOR 规则也使用了非规范化。请记住，如果是为了性能而做的修改，则必须在修改前后都进行性能测量。

- “规范化也就是说将数据移到子表中，然后使用伪键引用它们。”

  错！你可以为了方便、性能或者存储效益等所有合理的理由而使用伪键，但别认为这和规范化有什么关系。

- “规范化就是将属性尽可能地拆分开，就像 EAV 设计那样。”

  错！通常程序员会误解“规范化”这个词，认为它把数据弄得更不可读或者不便于查询。而事实上，真正的“规范化”正好与之相反。

- “没人需要超过第三范式的规范化标准。其他的范式都太晦涩难懂，并且你永远不会用到它们的。”

  错！调查表明，超过 20%的商业数据库的设计满足第一到第三范式，但违背第四范式。虽然这个量看上去比较小，但并不能说它就可以忽略：如果 20%的程序中存在潜在的 Bug 会导致数据丢失，你会不想解决它吗？

A.3 什么是规范化

下面这些是规范化的目标：

- 以一种我们能够理解的方式表达这个世界中的事物；
- 减少数据的冗余存储，防止异常或者不一致的数据；
- 支持完整性约束。

请注意，提高数据的性能并不在此列表中。规范化有助于我们正确地存储数据，避免陷入麻烦。当数据库是非规范化的时候，几乎不可避免地会变成一个烂摊子，我们需要编写更多的代码来清理不一致的或者重复的数据，最终我们会因为错误数据而不得不延期项目以及投入更多的成本。如果将所有这些场景考虑进去，就更容易看出规范化所带来的效率提升。

当一张表满足规范化的规则时，我们便称这张表符合*范式*。有五种传统的范式，描述了依次递进的规范化等级。每种范式消除了一种特定类型的冗余或者异常情况。通常来说，如果一张表的设计满足某一层级的范式，那这张表一定满足前面所有层级的范式。研究人员还定义了这五种传统范式之外的另外三种范式。范式的渐进级别如图 A-1 所示。

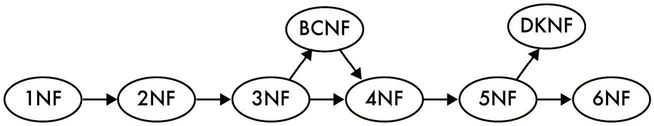


图 A-1 范式渐进级别

A.3.1 第一范式

第一范式的最根本要求是，该表必须是一个关系。如果它不符合 A.1 节中所描述的关系准则，那么这张表就不符合第一范式和后续的范式。

接下来的要求是这张表必须没有重复组合。请记住，关系中的每一行，都是从多个集合的每一个集合中选一个值形成的一种组合。重复的组合说明这一行可能有多个来自于同一个集合的值。

我们曾经见过两个创建了重复组合的反模式。

- 第 8 章在多个列中出现了来自同一个域的值。
- 第 2 章在同一列中有多个值。

从图 A-2 中可以看到这两个反模式中的重复组合。满足第一范式的更合理的设计应该是创建一个单独的表，tag 占用单独的一列，并且通过每行存储一个标签来支持多个标签的存储。

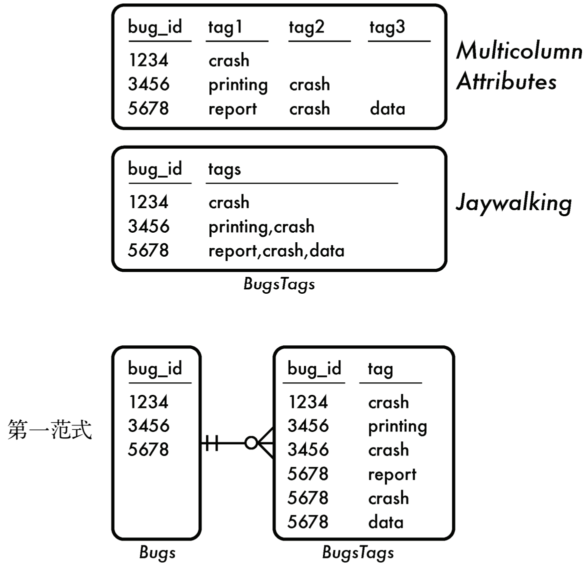


图 A-2　重复组合与第一范式

A.3.2　第二范式

除了复合主键之外，第二范式和第一范式是一样的。在之前的标签例子中，我们保持跟踪哪些用户给 Bug 打上了特定标签，我们还要关注是谁第一个创造了某一个标签。

Normalization/2NF-anti.sql

```
CREATE TABLE BugsTags (
  bug_id  BIGINT NOT NULL,
  tag     VARCHAR(20) NOT NULL,
  tagger  BIGINT NOT NULL,
  coiner  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (tagger) REFERENCES Accounts(account_id),
  FOREIGN KEY (coiner) REFERENCES Accounts(account_id)
);
```

在图 A-3 中，可以看到创造者标识的存储是冗余的。[①]这意味着修改了某一个 tag（如 crash）的创造者的标识，如果没有同步所有相同标签的行，则会导致数据异常。

① 此图使用用户名字而非 ID 号来标识用户。

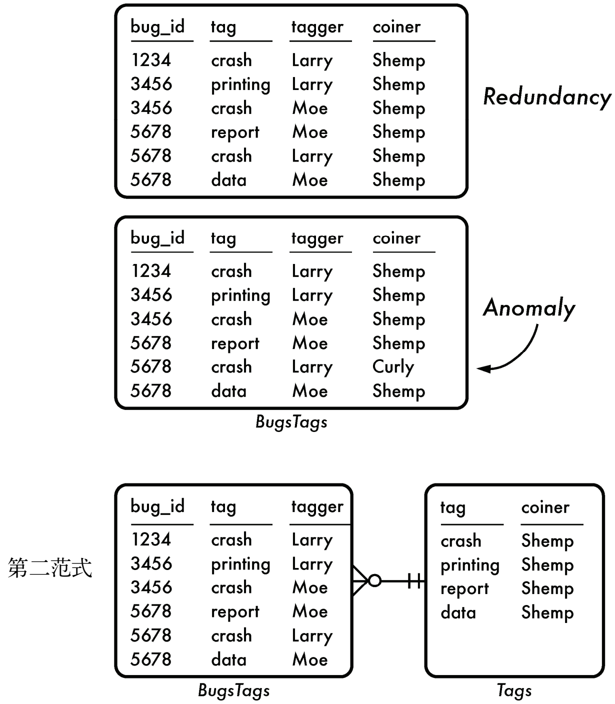


图 A-3　冗余与第二范式

为了满足第二范式，我们应该对于每个 tag 只存储一次它的创造者。也就是说，我们不得不额外定义一张表 Tags，以 tag 作为主键，这样每一个 tag 就只有一行了。接着，我们就可以将 tag 的创造者从 BugsTags 表里移到这张表中，从而防止了异常发生。

Normalization/2NF-normal.sql

```
CREATE TABLE Tags (
  tag     VARCHAR(20) PRIMARY KEY,
  coiner  BIGINT NOT NULL,
  FOREIGN KEY (coiner) REFERENCES Accounts(account_id)
);

CREATE TABLE BugsTags (
  bug_id  BIGINT NOT NULL,
  tag     VARCHAR(20) NOT NULL,
  tagger  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (tag) REFERENCES Tags(tag),
  FOREIGN KEY (tagger) REFERENCES Accounts(account_id)
);
```

A.3.3 第三范式

在 Bugs 这张表中，可能需要记录处理 Bug 的工程师的 E-mail。

Normalization/3NF-anti.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY
  -- . . .
  assigned_to BIGINT,
  assigned_email VARCHAR(100),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);
```

然而，E-mail 是这个被分配任务的工程师账号的一个属性，并不是 Bug 的属性。以这种形式存储 E-mail 就是冗余的，并且我们需要面对之前不满足第二范式时产生的那种风险。

第二范式例子中的那个犯规的列至少还部分和复合主键相关，而在第三范式的这个例子中，E-mail 这一列和主键就没有一点关联了。

要解决这一问题，我们需要将 E-mail 地址放到 Accounts 表中。图 A-4 显示了如何拆分 Bugs 表。由于在 Accounts 表中的 E-mail 是直接和主键关联而没有冗余的，因而这才是 E-mail 的合适位置。

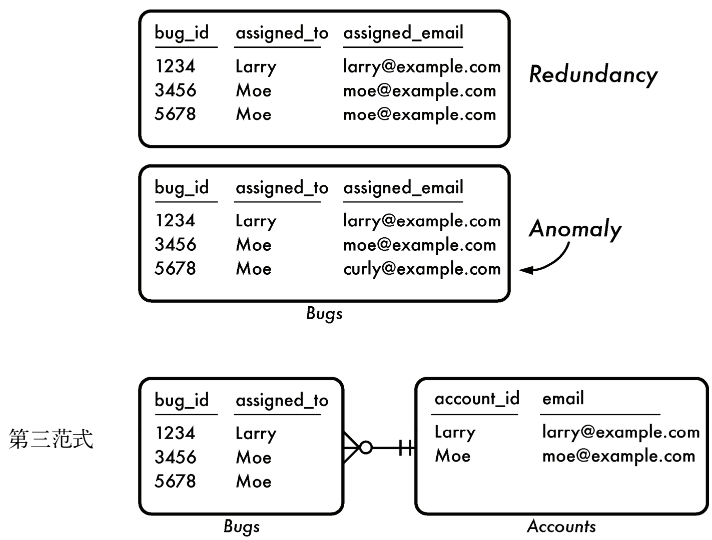


图 A-4 冗余与第三范式

A.3.4 博伊斯–科德范式

比第三范式稍微严格一点的范式版本称为博伊斯–科德范式。这两个范式之间的不同之处在

于，在第三范式中，所有的非关键字列都必须直接依赖于这张表中的关键字列，而在博伊斯–科德范式中，所有关键字列也必须遵循这一规则，这一点在一张表有多种列的集合可作为表的关键字时才有效。

比如，我们有三种 `Tag` 类型：描述 Bug 所造成影响的 tag，描述 Bug 影响子系统的 tag，以及 Bug 修复状态的 tag。每一个 Bug 对于每一种 `Tag` 类型只能有一个 tag。可能的键组合包括 `bug_id` 加上 `tag`，或者 `bug_id` 加上 `tag_type`。这两种组合都应该足够定位每一个独立的行。

在图 A-5 中，可以看到一个满足第三范式但是不满足博伊斯–科德范式的表，以及如何修改才能让它满足博伊斯–科德范式。

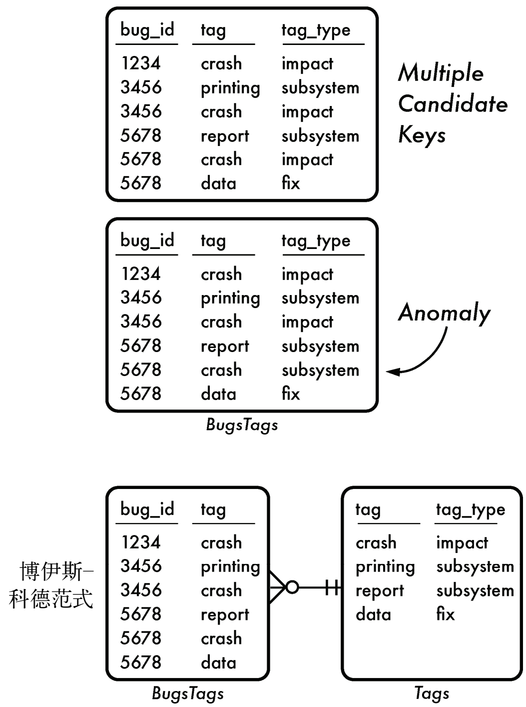


图 A-5　第三范式与博伊斯–科德范式

A.3.5　第四范式

现在，我们需要修改数据库，以支持多个用户报告同一个 Bug，并分配给多个开发工程师，然后由多个质量工程师验证。我们知道多对多的关系需要一张额外的表：

Normalization/4NF-anti.sql

```
CREATE TABLE BugsAccounts (
```

```
  bug_id       BIGINT NOT NULL,
  reported_by  BIGINT,
  assigned_to  BIGINT,
  verified_by  BIGINT,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id)
);
```

我们不能单独使用 `bug_id` 作为主键。每个 Bug 需要多行数据来实现各个字段都支持多个账号的目的。也不能将前两列或者前三列作为复合主键，因为这样，最后一列依旧不支持多个值。因此，主键必须是由所有四列组成。然而，`assigned_to` 和 `verified_by` 需要可以为空，因为 Bug 在被指派和验证之前是可以报告的。而标准情况下，所有的主键列都有一个非空的约束。

另一问题就是当某一列的账号数小于其他列时，就会造成数据冗余。图 A-6 显示了这种冗余。

bug_id	reported_by	assigned_to	verified_by
1234	Zeppo	NULL	NULL
3456	Chico	Groucho	Harpo
3456	Chico	Spalding	Harpo
5678	Chico	Groucho	NULL
5678	Zeppo	Groucho	NULL
5678	Gummo	Groucho	NULL

BugsAccounts

Redundancy,
NULLs,
No Primary Key

第四范式

bug_id	reported_by
1234	Zeppo
3456	Chico
5678	Chico
5678	Zeppo
5678	Gummo

BugsReported

bug_id	assigned_to
3456	Groucho
3456	Spalding
5678	Groucho

BugsAssigned

bug_id	verified_by
3456	Harpo

BugsVerified

图 A-6 合并关系与第四范式

上面提出的这些问题都是由于创建了一张做了双倍或者三倍工作的交叉表。当尝试使用一张交叉表描述多个多对多关系时，就会违背第四范式。

图 A-6 展示了我们可以拆分这张表来解决问题。我们应该为每一种多对多关系使用一张单独的交叉表，这就解决了冗余和数量不匹配的问题。

Normalization/4NF-normal.sql

```
CREATE TABLE BugsReported (
  bug_id       BIGINT NOT NULL,
  reported_by  BIGINT NOT NULL,
```

```
  PRIMARY KEY (bug_id, reported_by),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
);
CREATE TABLE BugsAssigned (
  bug_id       BIGINT NOT NULL,
  assigned_to  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);

CREATE TABLE BugsVerified (
  bug_id       BIGINT NOT NULL,
  verified_by  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, verified_by),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id)
);
```

A.3.6 第五范式

任何满足博伊斯–科德范式并且没有复合主键的表将同时满足第五范式。我们可以通过下面的这个例子来更好地理解第五范式。

有些工程师只为几个产品服务。我们应该将数据库设计成让我们了解谁在为哪些产品服务，以及在修复哪些 Bug，并且最小化数据的冗余。首先我们会想到在 BugsAssigned 表中增加一列来展示是哪个工程师在处理：

Normalization/5NF-anti.sql

```
CREATE TABLE BugsAssigned (
  bug_id       BIGINT NOT NULL,
  assigned_to  BIGINT NOT NULL,
  product_id   BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

但这不足以告诉我们这个工程师可以被指派为哪些产品服务，它只表明了这个工程师当前被指派去服务哪些产品，同时，这么做也造成了重复的存储。这是由于想在一张表中存储多种独立的多对多关系而产生的，就像我们在第四范式中所看到的问题一样。图 A-7 描述了这种冗余。[①]

① 图中使用了用户名字而不是 ID。

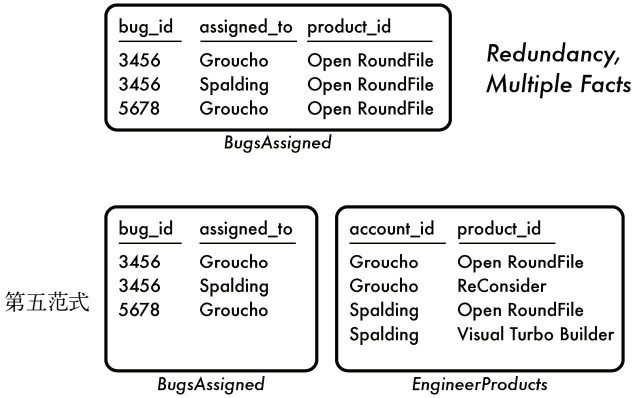


图 A-7 合并关系与第五范式

我们的解决方案是将每个关系都分别放到不同的表中：

Normalization/5NF-normal.sql

```
CREATE TABLE BugsAssigned (
  bug_id       BIGINT NOT NULL,
  assigned_to  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE EngineerProducts (
  account_id   BIGINT NOT NULL,
  product_id   BIGINT NOT NULL,
  PRIMARY KEY (account_id, product_id),
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

现在我们可以记录某个工程师能为哪个产品服务，而不依赖于这个工程师是否正在为那个产品的 Bug 努力。

A.3.7 更多的范式

DK 范式（Domain-Key normal form）认为表上的每个约束都是这张表的数据域约束和关键字约束的逻辑结果。DK 范式涵盖了第三、四、五范式和博伊斯–科德范式。

比如说，一个状态为 NEW 或者 DUPLICATE 的 Bug 应该是没有任何工作量的，因此应该没有工作时间记录，也不需要在 verified_by 列中指派质量工程师。可能的实现方法是使用一个触发器或者一个 CHECK 约束。这些都是建在表的非关键字列上的约束，因而它们并不符合 DK 范式

的标准。

第六范式旨在消除所有的联结依赖，它通常用来支持属性的变更历史。比如，我们可能想要在一张子表中记录下 Bugs.status 随着时间推移产生的变化：何时发生的变更，谁做的变更，以及其他可能的细节。

可以想象，如果让 Bugs 这张表满足第六范式，几乎每一列都需要附带一个历史记录表。这会导致表的数量过多。第六范式对于大多数程序来说都是没有必要的，但一些数据仓库技术里会使用到第六范式。①

A.4 常识

规范化规则并不深奥或者复杂。它们只是减少冗余和提高数据一致性的惯用技术方法。

你可以使用这份关于关系和范式的简单参考资料，来帮助自己在未来的项目中更好地设计数据库。

① 比如，Anchor Modeling 就是用了第六范式（http://www.anchormodeling.com）。

附录 B

参考书目

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns*. John Wiley and Sons, Inc., New York, 1998.

[Cel04] Joe Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann Publishers, San Francisco, 2004.

[Cel05] Joe Celko. *Joe Celko's SQL Programming Style*. Morgan Kaufmann Publishers, San Francisco, 2005.

[Cod70] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Reading, MA, first edition, 2003.

[Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley Longman, Reading, MA, 2003.

[Gla92] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, Reading, MA, 1992.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, pages 5–48, March 1991. Reprinted http://www.validlab.com/goldberg/paper.pdf.

[GP03] Peter Gulutzan and Trudy Pelzer. *SQL Performance Tuning*. Addison-Wesley, 2003.

[HLV05] Michael Howard, David LeBlanc, and John Viega. *19 Deadly Sins of Software Security*. McGraw-Hill, Emeryville, California, 2005.

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer:From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.

[Lar04] Craig Larman. *Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, Englewood Cliffs, NJ, third edition, 2004.

[RTH08] Sam Ruby, David Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, third edition, 2008.

[Spo02] Joel Spolsky. The law of leaky abstractions. http://www.joelonsoftware.com/articles/LeakyAbstractions.html, 2002.

[SZT+08] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy Zawodny, Arjen Lentz, and Derek J. Balling. *High Performance MySQL*. O'Reilly Media, Inc., second edition, 2008.

[Tro06] Vadim Tropashko. *SQL Design Patterns*. Rampant Techpress, Kittrell, NC, USA, 2006.

SQL Antipatterns Avoiding the Pitfalls of Database Programming

SQL反模式

多数软件开发人员并不是SQL专家，很多人对SQL的错误使用更使其效率低且难以维护。本书针对SQL使用中经常犯的错误展开分析，从数据库的逻辑设计、物理设计、查询设计、应用开发几个方面总结归纳各种典型错误，提出避免陷阱的方法。作为一本经验总结性的著作，本书是数据库编程人员不可或缺的手边书。你也会学到最新的全文搜索技术，设计出可以防范SQL注入的代码，掌握其他非常实用的使用技巧。

"我是最佳实践的最坚定拥护者，因为我喜欢从别人的错误中吸取教训。这本书广泛收集人们犯过的错误，令我吃惊的是，有些也是我犯过的。我真后悔没有早点读这本书。"

——Marcus Adams，资深软件工程师

"比尔写的是一本引人入胜、实用、重要而独一无二的书。书中描述的反模式与解决方案让软件开发人员实实在在地受益，我马上就使用了书中的技巧改善了我的应用程序。了不起的作品！"

——Frederic Daoud，

*Stripes: ...And Java Web Development Is Fun Again*与*Getting Started with Apache Click*的作者

"很明显，本书是经年累月的SQL数据库实践经验的结晶，书中每一个话题的深度与对细节的把握远超出我的预期。虽然本书不是为初学者而写，但是任何有一定SQL经验的开发人员都会发现这是一本有价值的参考书，都能从中发现新的收获。"

————Mike Naberezny，

Maintainable Software合伙人, *Rails for PHP Developers*作者之一

"书中满是非常实用的建议，出版时机也恰好。当大家都在关注看起来不错的新玩意时，专业人员刚好有机会用本书提升他们的SQL功力。"

——Maik Schmidt，

Enterprise Recipes with Ruby and Rails 和*Enterprise Integration with Ruby*作者

The Pragmatic Programmers

图灵社区：www.ituring.com.cn
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/数据库

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-26127-4

9 787115 261274 >

ISBN 978-7-115-26127-4
定价:59.00元